

The Quest Language and System

Luca Cardelli

Digital Equipment Corporation, Systems Research Center
130 Lytton Avenue, Palo Alto CA 94301
luca@src.dec.com

Contents

1. Introduction	3
2. System tutorial	4
3. The big picture	8
3.1. Kinds, types, and values	8
3.2. Signatures and bindings	11
4. Language overview	14
5. The kind of types	17
5.1. Basic and built-in types	17
5.2. Function types	19
5.3. Tuple types	22
5.4. Option types	25
5.5. Auto and Dynamic types	27
5.6. Recursive types	29
5.7. Mutable types	30
5.8. Exception types	34
6. Operator kinds	35
6.1. Type operators	35
6.2. Recursive type operators	36
7. Power kinds	37
7.1. Tuple subtypes	39
7.2. Option subtypes	40
7.3. Function subtypes	41
7.4. Bounded universals	41
7.5. Bounded existentials	42
7.6. Auto subtypes	42
7.7. Var, Out, and Array subtypes	43
7.8. Recursive subtypes	45
8. Modularity	46
8.1. Interfaces and modules	46
8.2. Manifest types and kinds	48
8.3. Diamond import	49
8.4. Soundness	50
9. Acknowledgments	51
10. Appendix	52
10.1 Syntax	52
10.2 Type rules	56
10.3 Quest Machine	60
10.4 Library interfaces	65
References	72

1. Introduction

This draft has many purposes. The principal one is to serve as working documentation for the Quest project, which involves investigations in type theory, typechecking algorithms, compilation techniques, and run-time systems. The main components of this document are, or will be, the following.








- An introduction to higher-order type system.
- A reference manual for the Quest programming language.
- A tutorial for using the Quest system.
- A description of the implementation.
- Documentation for libraries and utilities.

This is not intended to be a terse reference manual or a formal specification of the language. Many details are left to the intuition of the reader. We hope this will improve over time.

This document is supposed to reflect the true state of the system as implemented, except where explicitly noted. If not, please let me know.

Paragraph icons

The following markers are used as guides through the sea of paragraphs.

-  If you read this, you will learn about a temporary inconsistency in the manual, or a temporary restriction in the implemented system. Eventually, these paragraphs will go away.
-  If you read this, you will learn additional details about some topic. If you do not, it should not hurt you too badly.
-  If you read this, you will learn some detail that you probably need to know in order to use the system.
-  If you read this, you will learn some detail that will help you making a better use of the language. If you do not, you may miss good opportunities.
-  This marks an exercise for the reader (no answers provided).
-  This indicates input given to the system.
-  This indicates output produced by the system.

2. System tutorial

The Quest language is implemented as an interactive compiler. This means that the system behaves very much like an interpreter for the unwitting user. The user inputs a phrase in order to evaluate an expression or to establish a binding; then the system produces an answer. In between, the system parses, statically scopes, typechecks, translates and executes the request.

Large Quest programs are kept in files, and are normally organized in interfaces and modules. These components can be separately compiled, then linked, and even saved as stand-alone programs. All this can be done interactively, or without ever entering the interactive environment. In the latter mode of usage, the system behaves more like a traditional compiler and linker.

This dual implementation approach has influence on the language design; Quest must strike a compromise between being a good interactive language for small programs, and a good batch language for large programs. The two view complement each other. Any complete interaction typescript is a legal Quest program, and can be saved and converted to a module (with minor syntactic changes). Conversely, every interface and module can be entered interactively.

The level of interaction at which a user enters a phrase and receives an answer is called the top level of the system.

☛ One reaches the top-level by typing “Quest” at a shell (assuming everything is set up properly).

The following notation is used to illustrate the top-level interaction: user input is preceded by a “☛” sign ; system output is preceded by a “☞” sign. The output portions are occasionally omitted when they are obvious.

Once at the top-level, we can for example bind a value to a variable, and then use that variable in an expression.

☛ User input is always terminated by a semicolon.)

```
☛ let a = 3;  
☞ let a: Int = 3  
  
☛ a+4;  
☞ 7 : Int
```

The keyword `let` is used for introducing value variables, other forms of bindings will be explained as they arise.

As a general convention, lower-case keywords and identifiers (possibly with internal capitalization) relate to values, while capitalized keywords and identifiers, such as `Int`, relate to types.

Simple functions are defined and used as follows, where `let rec` must be used for recursive functions. Lexical and syntactic details are explained later, but many simple facts can be guessed by examining these examples.

```

✎ let average(a:Int b:Int):Int = {a+b}/2;
➡ let average:All(a:Int b:Int):Int = <fun>
✎ average(3 5);
➡ 4 : Int

✎ let rec fact(n:Int):Int =
    if n is 0 then 1 else n*fact(n-1) end;
➡ let fact:All(n:Int):Int = <fun>
✎ fact(5);
➡ 120 : Int

```

Tuples of values with named components can be formed as follows:

```

✎ let t =
    tuple
        let a = 3
        let b = true
    end;
➡ let t : Tuple a:Int b:Bool end Int =
    tuple let a = 3 let b = true end
✎ t.a;
➡ 3 : Int

```

☛ Sequences of Quest top-level phrases can be stored in files that by convention have extension “.qst”. Suppose “foo.qst” contains the first two phrases in this section, then loading that file will produce the corresponding answers:

```

✎ load "foo.qst";
➡ let a:Int = 3
    7 : Int

```

Loadable files can contain arbitrary top-level commands, including load. Files of top-level commands should be used only temporarily or not at all. Quest source files normally contain interfaces and modules.

Interfaces and modules are the approved way of organizing Quest programs. A new interface named A and prescribing two variables x and y can be entered at the top-level by typing the following (interface names, being type names, are by convention capitalized):

```
✎ interface A
  export
    x:Bool
    y:Int
  end;
➡ (saving interface A)
```

The interface is compiled and saved. A module implementing this interface can then be entered by typing the following (module names, being value names, are by convention lower-case):

```
✎ module a:A
  export
    let x = true
    let y = 3
  end;
```

☞ The inner part of a module is a sequence of phrases exactly as they could be entered at the top level, except that they are not terminated by semicolons; a single semicolon terminates the whole module.

The effect of evaluating a module expression is that the relevant interfaces are loaded, then the module is compiled and saved.

```
➡ (loading interface A)
   (saving module a)
   (saving module info a)
```

Compiled interfaces and modules can be used in the same or a different session. To use a module at the top level one must link it into the system. This is achieved by importing the module at the top-level, along with its interface:

```
✎ import a:A
➡ (loading Interface A)
   (loading module a)
   (linking module a)
```

The relevant compiled components are loaded and linked. (Some of the above messages may be missing if the corresponding components are already loaded.)

The effect of this import statement is that an identifier called `a` is now defined at the top level, as well as a type identifier called `A`. Imported interfaces and modules are accessible as tuple types and tuples:

```

✎ :A;    (* What is the type A? *)
➡ :Tuple x:Bool y:Int end :: TYPE
✎ a;     (* What is the value a? *)
➡ tuple let x = true let y = 3 end : A
✎ a.x;   (* What is the x component of module a? *)
➡ true : Int

```

A more direct way to get access to the contents of a module is to open it; this is exactly like import, but in addition all the identifiers declared in the module interface become defined at the top level:

```

✎ open a:A
➡ let x:Bool = true
   let y:Int = 3
✎ x;
➡ true : Bool

```

Opening a module is sort of rude; it is provided only for interactive convenience, and is available only at the top level.

An import clause similar to the one shown above can be used in an interface or module to import other interfaces and modules. Interfaces and modules must be compiled after the interfaces they import, but not necessarily after the modules they import. If this rule is violated, a version-checking error is generated; for example if we redefine the interface A above:

```

✎ interface A
  export
    x:Int
    y:Bool
  end;
➡ (saving interface A)

```

and then we import module a without recompiling it, we get:

```

✎ import a:A
➡ Error: old module a imports new interface A

```

☞ The system maintains a load-set of loaded interfaces and modules, which contains at most one version of each. Attempting to import a component which is already loaded, will cause the existing version from the load-set to be used. Whenever an interface is recompiled, it and all the interfaces and modules that depend on it are flushed from the load-set. Similarly, when a module is recompiled, it and all the modules that depend on it are flushed.

☞ Interfaces and modules are usually stored in individual files, rather than typed at the top-level. Interfaces are stored in files of the same name as the interface and extension “.spec” Mod-

ules are stored in files of the same name as the module and extension “.impl”. Both are terminated by semicolons.

☛ Interfaces and modules stored in files can be compiled either by entering “Quest < file” at the shell, or by entering “load "file";” at the top-level. Compiled interfaces have extension “.spec.x” and compiled modules have extension “.impl.x”. Finally, the extension “.xtrn” is reserved for generic binary data.

☛ At start-up, the system loads a file called “Library.qst”. The contents of this file may change, but it normally contains commands to pre-link library interfaces (many of the ones listed in appendix), and commands to set system flags. Top-level definitions and load commands must not be added to this file. ☛ A different mechanism will be provided for user customization.

☛ The top level phrase “command"..";" gives access to a set of system flags. Type “command"";" for help, and do not use what you do not understand.

☛ Garbage collection is not yet implemented: compile large modules and interfaces in separate sessions, if needed.

3. The big picture

In this section we describe a very general type system characterized by a three-level structure of entities. This structure has driven many of the design decisions made in Quest.

If you find this section confusing, you should go directly to the next section to examine concrete examples.

3.1. Kinds, types, and values

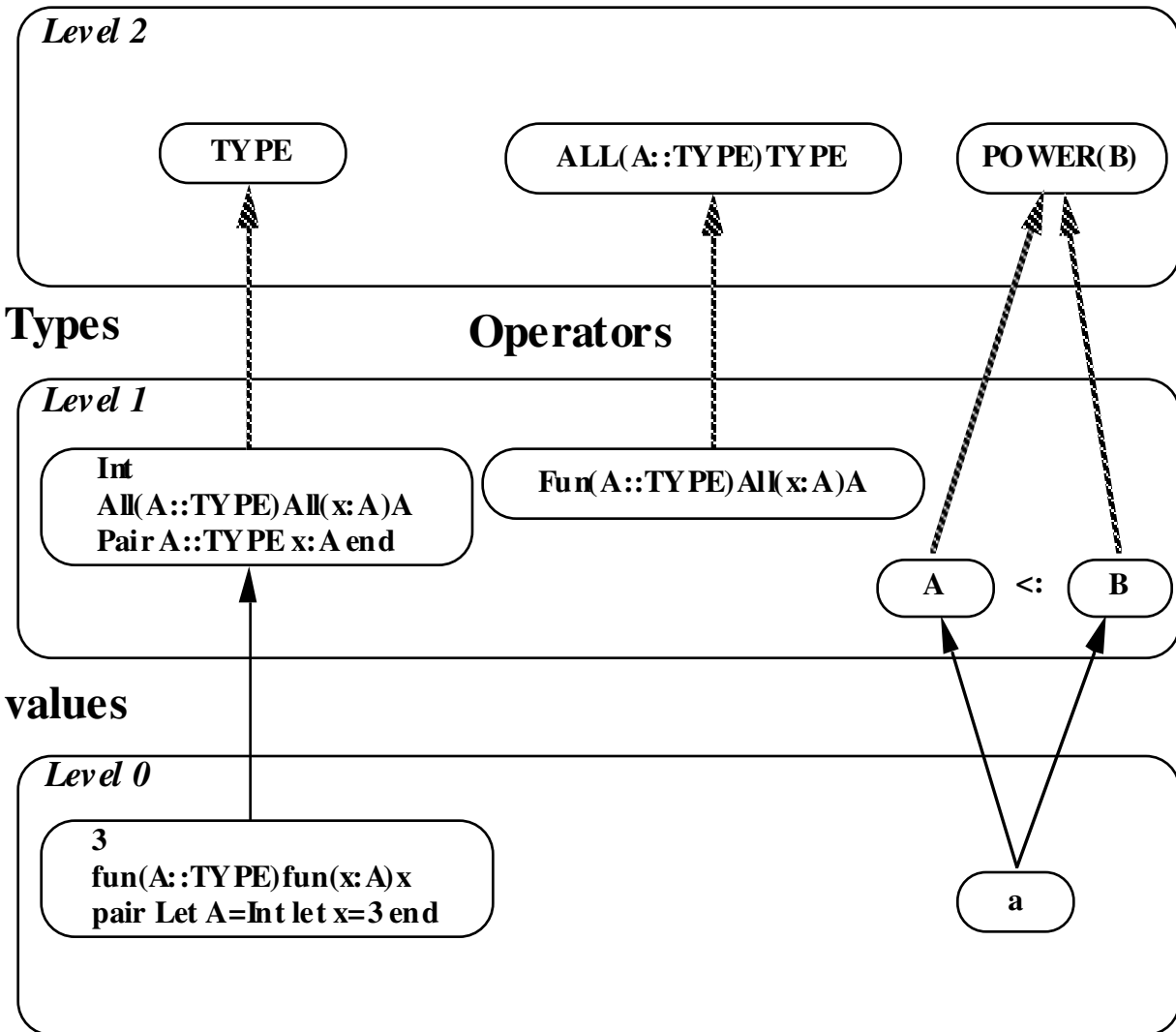
Ordinary languages distinguish between values and types. We can say that in this case there are two levels of entities; values inhabit Level 0, while types inhabit Level 1.

Quest adds a third level above types; this is called the level of kinds, or Level 2. Kinds are the “types” of types, that is they are collections of types, just like types are collections of values. The collection of all types is the kind TYPE (which is not a type). We use $a:A$ to say that a value a has a type A , and $A::K$ to say that a type A has a kind K .

In common languages, two levels are sufficient since the type level is relatively simple. Types are generated by ground types such as integers, and by operators such as function space and cartesian product. Moreover, there is normally only one (implicit) kind, the kind of all types, so that a general notion of kinds is not necessary.

Types in Quest have a much richer structure, including type variables, type operators, and a notion of type computation. In fact, the Quest type level is itself a very expressive λ -calculus, as opposed to a simple algebra of type operators. Moreover, this λ -calculus is “typed”, where the “types” here are the kinds, since we are one level “higher” than normal. This should become clearer by the end of this section.

KINDS



The level structure we adopt is shown in the level diagram. Values are at Level 0. At Level 1 we have types, intended as sets of values, and also type operators, intended as functions mapping types to types (e.g. the List type operator is a function mapping the type Int to the type of lists of integer). At Level 2 we have kinds which are the “types” of types and operators; intuitively kinds are either sets of types or sets of operators.

In more detail, at the value level we have basic values, such as 3; (higher-order) functions, such as `fun(x:A)x`; polymorphic functions, such as `fun(A::TYPE)fun(x:A)x`; pairs of values, such as `pair 3 true end`; and packages of types and values, such as `pair Let A=Int let x:A=3 end`.

Note that already at the value level types are intimately mixed with values, and not just as specifications of value components: polymorphic functions take types as arguments, and packages have types as components. This does not mean that types are first-class entities which can

play a role in value computations, but it does mean that types are not as neatly separated from values as one might expect.

At the type level we find, first of all, the types of values. So we have basic types, such as `Int`; function types, such as `All(x:A)A` (functions from `A` to `A`, usually written `A->A`); polymorphic types, such as `All(A::TYPE)All(x:A)A`; pair types, such as `Pair x:Int y:Bool end`; and abstract types, such as `Pair A::TYPE x:A end`.

At the type level we also find operators, which are functions from types to types (and from operators to operators, etc.) but which are not types themselves. For example `Fun(A::TYPE)All(x:A)A` is an operator that given a type `A` returns the type `All(x:A)A`. Another example is the parametric `List` operator, which given the type `Int` produces the type `List(Int)` of integer lists.

At the kind level we have the kind of all types, `TYPE`, and the kinds of operators, such as `ALL(A::TYPE)TYPE`, which is the kind of `List`.

The right-hand side of the level diagram illustrates subtyping. At the type level we have a subtyping relation, written `A<:B` (`A` is a subtype of `B`); if a value has type `A`, then it also has type `B`. At the kind level we have the `POWER` kinds for subtyping; if `B` is a type, then `POWER(B)` is the kind of all subtypes of `B`, in particular we have `B::POWER(B)`. Moreover, if `A<:B` then `A::POWER(B)`, and we identify these two notions.

We also have a relation of subkind, written `<::` (never actually used in programs), such that if `A<:B` then `POWER(A)<::POWER(B)`, and also `POWER(A)<::TYPE` (the latter means that the collection of subtypes of `A` is contained in the collection of all types).

A structure such as the one in the level diagram is characterized by the quantifiers it incorporates. (These are deeply related to the quantifiers of logic [Martin-Löf 80], but we shall not discuss this connection here.) A quantifier is intended as a type construction that binds variables. There are four possible universal quantifiers (classifying function-like objects) and four possible existential quantifiers (classifying pair-like objects). These basic quantifiers are binary: they involve a variable ranging over a first type or kind in the scope of a second type or kind. The number four above comes from all the possible type/kind combinations of such binary operators; the following examples should make this clear.

In our level structure we have three universal quantifiers (two flavors of `All` plus `ALL`), and two existential quantifiers (two flavors of `Pair`, of which only one appears in the diagram):

- `All(x:A)B`

This is the type of functions from values in `A` to values in `B`, where `A` and `B` are types. The variable `x` can appear in `B` only in special circumstances, so this is normally equivalent to the function space `A->B`. Sample element: `fun(x:Int)x`.

- `All(X::K)B`

This is the type of functions from types in `K` to values in `B`, where `K` is a kind, `B` is a type, and `X` may occur in `B`. Sample element: `fun(A::TYPE)fun(x:A)x`.

- ALL(X::K)L

This is the kind of functions from types in K to types in L, where K and L are kinds, and X may occur in L. Sample element: Fun(A::TYPE)A.

- Pair x:A y:B end

This is the type of pairs of values in A and values in B, where A and B are types. The variable x can appear in B only in special circumstances, so this is normally equivalent to the cartesian product A#B. Sample element: pair let x=true let y=3 end.

- Pair X::K y:B end:

This is the type of pairs of types in K and values in B, where K is a kind, B is a type, and X may occur in B. Sample element: pair let X=Int let y:X=3 end.

For symmetry, we could add a third existential quantifier PAIR X::K Y::L end (a kind), but this turns out not to be very useful. There are very specific reasons for not admitting the other two possible quantifiers (the kind of functions from values to types, and the kind of pairs of values and types, where the types may depend on the values) or the unrestricted forms of All(x:A)B and Pair x:A y:B end where x may occur in B. These quantifiers make compilation very problematic, and they are very regretfully but deliberately omitted.

Before proceeding, we should summarize and clarify our notation.

NOTATION

- We use lower case identifiers (with internal capitalization for compound names) for Level 0, capitalized identifiers for Level 1, and all caps for Level 2. For example, let introduces values, Let introduces types, and DEF introduces kinds.

- We use the Courier font for programs, Courier Bold for some program keywords, and Courier Italic for program meta-variables. In particular: x,y,z range over value variables; a,b,c range over value terms; X,Y,Z range over type (or operator) variables; A,B,C range over type (or operator) terms; U,V,W range over kind variables; K,L,M range over kind terms. Note that this does not apply to program (non-meta) variables: for example, we often use A (Roman) as a type variable.

- We use a:A to say that a value a has a type A; A::K to say that a type A has a kind K; A<:B to say that A is a subtype of B; and K<::L to say that K is a subkind of L.

These conventions have already been used implicitly in this section, and will be used heavily hereafter.

3.2. Signatures and bindings

The quantifiers we have examined in the previous section are all binary; they compose two types or kinds to produce a new type or kind. In programming languages it is much more convenient to use n-ary quantifiers, so that we can easily express functions of n arguments and tuples

of n components. N -ary quantifiers are achieved through the notions of signatures and bindings [Burstall Lampson 84].

A signature is a (possibly empty) ordered association of kinds to type variables, and of types to value variables, where all the variables have distinct names. For example:

$A::\text{TYPE} \quad a:A \quad f:\text{All}(x:A)\text{Int}$

This is a signature declaring a type variable, a value variable of that type, and a value variable denoting a function from objects of that type to integer. Note that signatures introduce variables from left to right, and such variables can be mentioned after their introduction.

A binding is a (possibly empty) ordered association of types to type variables and values to value variables, where all the variables have distinct names, for example:

Let $A::\text{TYPE} = \text{Int}$
 let $a:A = 3$
 let $f:\text{All}(x:A)\text{Int} = \text{fun}(x:\text{Int})x+1$

This is a binding associating the type Int and the values 3 and $\text{fun}(x:\text{Int})x+1$ with the appropriate variables. Bindings introduce variables from left to right; for example a could be used in the body of f . In some cases it is desirable to omit the variables and their related type information; the binding above then reduces to:

$:\text{Int} \quad 3 \quad \text{fun}(x:\text{Int})x+1$

The colon in front of Int is to tell a parser, or a human, that a type is about to follow. Such prefix colons are found in contexts where either a type or a value is expected; they are not needed where only types are expected.

We can now convert our binary quantifiers to n -ary quantifiers as follows, where S is a signature and A is a type (and pairs become tuples):

n -ary universals:	$\text{All}(S)A$
n -ary existentials:	$\text{Tuple } S \text{ end}$

NOTATION

- We use S as a meta-variable ranging over signatures, and D as a meta-variable ranging over bindings.
- The following abbreviations will be used extensively for components of signatures and bindings, where S_i are signatures, L is a kind, B is a type and b is a value:

Signatures:

$x_1, \dots, x_n:B$	for	$x_1:B..x_n:B$
$X_1, \dots, X_n::L$	for	$X_1::L..X_n::L$
$x(S_1)..(S_n):B$	for	$x:\text{All}(S_1).. \text{All}(S_n)B$
$X(S_1)..(S_n)::L$	for	$X::\text{ALL}(S_1).. \text{ALL}(S_n)L$

Bindings:

let <u>x(S₁)..(S_n):B=b</u>	for	let x=fun(S ₁)..fun(S _n):B b
Let X(S ₁)..(S _n):L=B	for	Let X=Fun(S ₁)..Fun(S _n):L B

Signatures and bindings can be used in many different places in a language, and hence make the syntax more uniform. The positions where signatures and bindings appear in a surrounding context are here shown underlined; note the use of some of the abbreviations just introduced:

Signatures in bindings:

Let A::TYPE=Int let a:A=3 let f(x:A):Int=x+1

Signatures in formal parameters:

let f(A::TYPE a:A f(x:A):Int):Int = ...
let f(_):Int = ...

Signatures in types:

All(A::TYPE a:A f(x:A):Int) A
All(_) A
Tuple A::TYPE a:A f(x:A):Int end
Tuple _ end

Signatures in interfaces:

interface I
import m1:I1 m2:I2
export
 A::TYPE
 a:A
 f(x:A):Int
end

Bindings at the top-level:

Let A::TYPE=Int; let a:A=3; let f(x:A):Int=x+1;
:Int; 3; fun(x:Int) x+1;
_;

Bindings in actual parameters:

f(Let A::TYPE=Int let a:A=3 let f(x:A):Int=x+1)
f(:Int 3 fun(x:Int) x+1)
f(_)

Bindings in tuples:

tuple Let A::TYPE=Int let a:A=3 let f(x:A):Int=x+1 end
tuple :Int 3 fun(x:Int) x+1 end
tuple _ end

Bindings in modules:

```
module m:I
import m1:I1 m2:I2
export
  Let A::TYPE=Int
  let a:A=3
  let f(x:A):Int=x+1
end
```

Interfaces and modules will be discussed later.

4. Language overview

The design principles and ideas we have discussed so far, as well as some yet to be discussed, are incorporated in the Quest programming language. Quest (a quasi-acronym for Quantifiers and Subtypes) illustrates many fundamental concepts of typing in programming, and how they can be concretely embedded and integrated in a single language.

The three major sections to follow deal with: a) the kind of types, including basic types, various structured types, polymorphism, abstract types, recursive types, dynamic types, mutable types and exception types; b) the kinds of operators, including parametric, recursive, and higher-order type operators; and c) the kinds of subtypes, including subtyping and bounded quantification. Then follow two sections on programming with modules and interfaces, seen as values and types respectively, and a short section on system programming from the point of view of typing violations.

The main unifying concepts in Quest are those of type quantification and subtypes. However, a number of important type constructions do not fall neatly in these classes (e.g. mutable type and exceptions types), and they are added on the side. The important point is that they do not perturb the existing structure too much.

Here is a brief, general, and allusive overview of Quest characteristics; a gradual introduction begins in the next section.

☞ Quest has three levels of entities: 1) values, 2) types and operators, 3) kinds. Types classify values, and kinds classify types and type operators. Kinds are needed because the type level is unusually rich.

☞ Explicit type quantification (universal and existential) encompasses parametric polymorphism and abstract types. Quantification is possible not just over types (as in ordinary polymorphic languages) but also over type operators and over the subtypes of a given type.

☞ Subtyping is defined inductively on all type constructions (e.g. including higher-order functions and abstract types). Subtyping on tuple types, whose components are ordered, provides a form of single inheritance by allowing a tuple to be viewed as a truncated version of itself with fewer final components.

☞ There are user-definable higher-order type operators and computations at the type level. The typechecker includes a normal-order typed λ -calculus evaluator (where the "types" here are actually the kinds).

☞ A generalized correspondence principle is adopted, based on the notions of signatures and bindings. Landin proposed a correspondence between declarations and formal parameters. Burstall and Lampson proposed a correspondence between interfaces and declarations, and between parametric modules and functions [Burstall Lampson 84]. In Quest there is a correspondence between declarations, formal parameters, and interfaces, all based on a common syntax, and between definitions, actual parameters, and modules, also based on a common syntax.

☞ Evaluation is deterministic, left-to-right, applicative-order. That is, functions are evaluated before their arguments, the arguments are evaluated left to right, and the function bodies are evaluated after their arguments. In records and tuples, the components are evaluated left to right, etc. Conditionals, cases, loops, etc. evaluate only what is determined by their particular flow of control. All entities are explicitly initialized.

☞ The general flavor of the language is that of an interactive, compiled, strongly-typed, applicative-order, expression-based language with first-class higher-order functions and imperative features. Type annotations are used rather heavily, in line with existing programming-in-the-large languages.

☞ In viewing signatures and bindings in terms of quantifiers, it is natural to expect alpha-conversion (the renaming of bound variables) to hold. Alpha-conversion would however allow signatures to match each other independently of the names of their components; this is routine for signatures in functions but very strange indeed for signatures in tuples, and it would create too many "accidental" matches in large software systems. Hence, Quest signatures and bindings must match by component name (and of course by component order and type). Some negative aspects of this choice are avoided by allowing component names in signatures and bindings to be omitted; omitted names match any name for the purpose of typechecking, and therefore provide a weak form of alpha-conversion.

☞ Limited type inference is provided. First of all, the result type of a function or expression is automatically deduced from the types of the components of such expression; for example the type of let-bound variables can be deduced. By limited type inference we mean that it is possible to omit some of the type information that normally would be required for type deduction (in the above sense). It works as follows. The type of each program variable must be explicitly provided (or be deduceable), and type parameters must be explicitly listed; such information is not subject to inference. However, type applications (instantiations of polymorphic functions to particular types) can "often" be omitted, when the context is sufficiently informative. For example, a polymorphic cons function can be used as `cons(:Int 3 tail)`, or (by inference from the types of 3 and tail) `cons(3 tail)`. Similarly; the polymorphic nil value can be used as `cons(3 nil(:Int))`, or (by inference from the type required by this instance of cons) `cons(3 nil())`. The parentheses after nil are still needed because nil by itself denotes a polymorphic constant. Note also that, in isolation, `nil()` does not express enough information to decide "which" nil we are talking about.

We conclude this section with some further notes about Quest syntax.

☞ Comments are enclosed in "(" and ")" and can be nested. A comment is lexically equivalent to a blank. Quotes (') enclose character literals, and double quotes (") enclose string literals.

☞ Curly brackets "{" and "}" are used for grouping value, type, and kind expressions (e.g. to force operator precedence). Parentheses "(" and ")" are used in formal and actual parameter lists, that is for function declarations, definitions and applications.

☞ The syntax has virtually no commas or semicolons. Commas are used only in lists of identifiers, and semicolons are used only to terminate top-level sentences, or whole modules. Otherwise, blank space is used. Indentation is not required for disambiguation, but it greatly improves readability because of the absence of normal separators. The resulting syntax is still easy to parse because of widespread use of initial and final keywords.

☞ There are two lexical classes of identifiers: alphanumeric (letters and numerals starting with a letter) and symbolic (built out of !@#\$%&* _+=-!\`:<>/?). Reserved keywords belonging to either lexical class are not considered identifiers. Alphanumeric identifiers are never used in infix position (e.g. $f(x\ y)$), but not $x\ f\ y$). Symbolic identifiers are always infix, but can also be used as prefix or stand-alone (e.g. $x+y$, $+(x\ y)$, or $\{+\}$).

☞ New infix operators can be declared freely (e.g. $\text{let } *+(x,y,z:\text{Int}):\text{Int} = x*\{y+z\}$).

☞ New listfix operators can be introduced. The primitive listfix operator is $\text{array of } a_1 \dots a_n \text{ end}$. In addition, any function of input signature $\text{Array}(A)$, e.g. $\text{sum}(\text{summands}:\text{Array}(\text{Int}))\text{Int}$, can be used as a listfix operator by writing $\text{sum of } a_1 \dots a_n \text{ end}$. More precisely, $f \text{ of } \dots \text{end}$ is read as an abbreviation for $f(\text{array of } \dots \text{end})$. By interaction with the type inference features, this works also for polymorphic functions of input signature $X::\text{TYPE } x:\text{Array}(X)$, etc.

☞ There is no declaration of infix status. (It is not needed because of the strict lexical conventions.) There is no declaration of infix precedence. All infix operators, including the built-in ones, have the same precedence and are right-associative. This decision has been made because it is difficult to specify the relative precedence of operators in a meaningful way, and because it is equally difficult to remember or guess the relative precedence of operators when new infixes can be introduced.

☞ There is no overloading of literals or operators, not even built-in ones. For example, 2 is an integer, 2.0 is a real, + is integer plus, and ++ is real plus.

☞ The style conventions we adopt are as follows. Value identifiers (including function names and module names) are in lower case, with internal capitalization for compound names. Type identifiers (including interface names) are capitalized. Kind identifiers are all caps. Keywords are capitalized in roughly the same way (e.g. "Tuple..end" is a tuple type, and "tuple..end" is a tuple value). Flow-control keywords (such as "begin..end" and "if..then..else..end") are in lower case. Keywords are pretty-printed in boldface, and comments in italics.

☞ The infix **:** sign means "has type". The infix **::** sign means "has kind". The infix **<:** sign means "is subtype of".

☛ The = sign is used to associate identifiers with values, types, and kinds. The = sign is not used as the boolean test for equality: the constructs `x is y` and `x isnot y` are used instead.

☛ The := sign is used as the assignment operator.

☛ If a module, e.g. `list`, implements a single type, or has a main type, that type is simply called `T`, as in `list.T`.

☛ Bug: open comments may span across file boundaries and leave you in a responsiveness state at the top level. In doubt, try `"*);"`.

5. The kind of types

The kind of types is the only kind that is supported by ordinary programming languages such as Pascal. Hence, this section will include many familiar programming constructs. However, we also find some relatively unusual programming features, namely polymorphism and abstract types in their full generality.

Basic types should cause no surprise. Function types use the general notion of signatures (universally quantified) to represent explicit polymorphism. Tuple types similarly use general signatures (existentially quantified) to represent abstract types. Another quantifier, summation of signatures over a finite set, accounts for the familiar disjoint union types. Infinite summation of signatures over types corresponds to dynamic typechecking and supports persistent data. Recursive types allow the construction of lists and trees. Mutable types (assignable variables and fields, and arrays) have to be handled carefully to maintain type soundness. Finally, exception types take care of exceptions.

5.1. Basic and built-in types

The basic types are `Ok`, `Bool`, `Char`, `String`, `Int`, `Real`, `Array`, and `Exception`. In a language with modules there is, in principle, no need for basic types because new basic types can be introduced by external interfaces (some of which may be known to the compiler, so that they are implemented efficiently). However, it is nice to have direct syntactic support for some commonly used constants and operators; this is the real reason for having special types designated "basic" (this need could perhaps be eliminated if the language supported syntactic extensions).

Types that are not basic but are defined in interfaces known to the compiler, are called built-in. The respective modules and interfaces are also called built-in. Examples of built-in interfaces are `Reader`, `Writer`, and `Dynamic`, described in the Appendix. Other built-in interfaces can also be defined as needed by implementations.

Ok

`Ok` is the type with a single constant `ok` and no operations. This type is sometimes called `void` or `unit` in other languages, and is used in expression-based languages as the type of expressions that are not really meant to have any value (e.g. assignment statements).

Bool

Bool is the type with constants true and false. Boolean operations are the infix \vee (or) and \wedge (and), and the function not; they evaluate all their arguments. The equality or inequality of two values is given by the forms `x is y` and `x isnot y`, which return a boolean. This is-ness predicate is defined as ordinary equality for Ok, Bool, Char, Int, and Real, and as "same memory location" for all the other types including String. Abstract types should always define their own equality operation in their interface.

A basic programming construct based on booleans is the conditional:

```
if true then 3 else 4 end;  
3 : Int
```

The conditional can take many forms. It may or may not have the then and else branches (missing branches have value ok and type Ok), and additional elsif branches can be cascaded.

Two other conditional constructs are provided:

<code>a andif b</code>	same as:	<code>if a then b else false end</code>
<code>a orif b</code>	same as:	<code>if a then true else b end</code>

These are useful to express complex boolean conditions when not all subconditions should be evaluated. For example, assuming `a`, `n`, and `m` are defined:

```
if {n isnot 0} andif {{m/n}>0} then a:=m  
  elsif {n is 0} orif {m<0} then a:=0  
end;  
ok : Ok
```

This conditional never causes division by zero.

Char

Char is the type of Ascii characters, such as 'a', which are syntactically enclosed in single quotes.

Backslash can be used to insert some special characters within character quotes: `\n` is newline, `\t` is tab, `\b` is backspace, `\f` is form feed, `\'` is single quote, and `\\` is backslash; backslash followed by any other character is equivalent to that character.

Operations on characters are provided through a built-in module `ascii` of built-in interface `Ascii` (this is written `ascii:Ascii`).

Incidentally, this is an example of how to organize basic operations into interfaces in order not to clutter the language: one might forget that `char` is an `Ascii` operation and confuse it with a variable or function, but on seeing `ascii.char` the meaning should be obvious.

String

String is the type of strings of Ascii characters, such as "abc", enclosed in double quotes.

☞ As with characters, backslash can be used to insert some special characters within string quotes: `\n` is newline, `\t` is tab, `\b` is backspace, `\f` is form feed, `\"` is double quote, and `\\` is backslash; backslash followed by any other character is equivalent to that character.

The operations on strings are provided through a built-in module `string:StringOp`, but there is also a predefined infix operator `<>` for string concatenation:

```
☞ let s1="concat" and s2="enate";  
☛ let s1:String = "concat"  
    let s2:String = "enate"  
☞ s1 <> s2;  
☛ "concatenate" : String
```

This example also illustrates simultaneous bindings, separated by `and`.

☞ Another library module `conv:Conv` contains routines for converting various quantities to strings; this is useful for printing.

Int

`Int` is the type of integer numbers. Negative numbers are prefixed by a tilde, like `~3`. The integer operators are the infix `+`, `-`, `*`, `/`, `%` (modulo), `<`, `>`, `<=`, and `>=`.

Real

☞ Beware, very little attention has been put into making reals really work.

`Real` is the type of floating point numbers. Real constants are formed by an integer, a dot, a positive integer, and optionally an exponent part, as in `3.2` or `~5.1E~4`.

☞ Exponential notation is not implemented.

☞ Reals are only single-precision in this implementation; this is not expected to change.

The real operators are the infix `++`, `--`, `**`, `//`, `^^` (exponent), `<<`, `>>`, `<<=`, and `>>=`.

```
☞ 2.8 ++ 3.4;  
☛ 6.2: Real
```

☞ There is no overloading between integer and real operations. Conversions and other operations are provided through the built-in module `real:RealOp`

☞ Exponent is not implemented. There is a built-in trigonometry interface `trig:Trig`, but not much of it is implemented.

Others

Arrays, exceptions, readers, writers, and dynamics are discussed in later sections, or in the Appendix.

5.2. Function types

Function types are normally realized, implicitly or explicitly, by an "arrow" operator requiring a domain and a range type. Here we adopt an operator (actually, a quantifier) requiring a do-

main signature; this is more general because the signature may introduce type variables, which can then occur in the range type.

A function type has the form:

$\text{All}(S)A$

This is the type of functions with parameters of signature S and result of type A :

$\text{fun}(S):A \rightarrow b$ or, abbreviated $\text{fun}(S)b$

Depending on the form of S and A , functions can be simple, higher order (if S or A contain function types), or polymorphic (if S introduces type variables).

☞ The result type A must not have any free value-variables that are bound in S .

Simple functions

Here is a simple function declaration:

```
☞ let succ = fun(a:Int):Int a+1;
☛ let succ(a:Int):Int = <fun>
☞ succ(3);
☛ 4 : Int
```

As discussed earlier, these declarations are usually abbreviated as follows:

```
☞ let succ(a:Int):Int = a+1;
☛ let succ(a:Int):Int = <fun>
```

Here is a function of two arguments; when applied it can take two simple arguments, or any binding of the correct form:

```
☞ let plus(a:Int b:Int):Int = a+b;
☛ let plus(a:Int b:Int):Int = <fun>
☞ plus(3 4);
☛ 7 : Int
☞ plus(let a=3 let b=a+1);
☛ 7 : Int
```

Recursive functions are defined by a `let rec` binding:







```
☞ let rec fact(n:Int):Int =
    if n is 0 then 1 else n*fact(n-1) end;
☛ let fact(n:Int):Int = <fun>
```

Simultaneous recursion is achieved by `let rec` followed by multiple bindings separated by `and`. All the entities in a recursive definition must syntactically be constructors, that is functions, tuples, or other explicit data structures, but not function applications, tuple selections, or other operations.


✂ The only constructors currently allowed are functions.

Higher-order functions

There is not much to say about higher-order functions, except that they are allowed:





```
 let double(f(a:Int):Int)(a:Int):Int = f(f(a));  
 let double(f(a:Int):Int)(a:Int):Int = <fun>  
  
 let succ2 = double(succ);  
 let succ2(a:Int):Int = <fun>  
  
 succ2(3);  
 5 : Int
```


Note that a function body may contain non-local identifiers (such as `f` in the body of `double`), and these identifiers are retrieved in the appropriate, statically determined scope.

 Define function composition for integer functions.



Polymorphic functions

Polymorphic functions are functions that have types as parameters, and that can later be instantiated to specific types. Here is the polymorphic identity function:





```
 let id(A::TYPE)(a:A):A = a;  
 let id(A::TYPE)(a:A):A = <fun>  
  
 id(:Int)(3);  
 3 : Int
```

 There is automatic inference of type parameters. (Expand.)





The identity function can be instantiated in many interesting ways. First, it can be instantiated to a simple type, like `Int` to obtain the integer identity:

```
 let intId = id(:Int);  
 let intId(a:Int):Int = <fun>
```

It can be applied to the integer identity by first providing its type, the integer endomorphism type:

```
 Let IntEndo = All(a:Int)Int;  
 Let IntEndo::TYPE = All(a:Int)Int  
  
 id(:IntEndo)(intId);  
 <fun> : All(a:Int)Int
```



 It can even be applied to itself by first providing its own type:

```
 Let Endo = All(A::TYPE)All(a:A)A;  
 Let Endo::TYPE = All(A::TYPE)All(a:A)A  
  
 id(:Endo)(id);  
 <fun> : All(A::TYPE)All(a:A)A
```



Type systems that allow polymorphic functions to be applied to their own types are called impredicative. The semantics of such systems is delicate, but no paradox is necessarily involved.


 Define polymorphic function composition.

Polymorphic functions are often curried in their type parameter, as we have done so far, so that they can be conveniently instantiated. However, this is not a requirement; we could also define the polymorphic identity as follows:

```
 let id2(A::TYPE a:A):A = a;  
➡ let id2(A::TYPE a:A):A = <fun>  
 id2(:Int 3);  
➡ 3 : Int
```

A slightly more interesting polymorphic function is twice, which can double the effect of any function (including itself):

```
 let twice(A::TYPE)(f(a:A):A)(a:A):A = f(f(a));  
➡ let twice(A::TYPE)(f(a:A):A)(a:A):A = <fun>  
 let succ2 = twice(:IntEndo)(succ);  
➡ let succ2(a:Int):Int = <fun>
```

 Write a typed version of the untyped term $d2 = \text{fun}(x)x(x)$, by inserting additional type information where needed. Can one do this for $d3 = \text{fun}(x)x(x(x))$? What about $d2(d2)$?

5.3. Tuple types

Tuple types are normally intended as iterated applications of the cartesian product operator. Our tuple types are formed from signatures; this is more general because signatures can introduce type variables which may occur in the following signature components. These tuple types can be interpreted as iterated existential quantifiers.


A tuple type has the form:

Tuple S end

This is the type of tuples containing bindings D of signature S:

tuple D end

Depending on the form of S, we can have a simple tuple type or an abstract type (if S introduces type variables).

 Value-variables declared in S may appear in the types and kinds of S. Note that these variables are not free in S.

Simple tuples

A simple tuple is an ordered collection of values. These values can be provided without names:

```

✎ tuple 3 4 "five" end;
⇒ tuple 3 4 "five" end : Tuple :Int :Int :String end

```

Then one way to extract such values is to pass the tuple to a function that provides names for the components in its input signature. But frequently, tuple components are named:

```

✎ let t: Tuple a:Int b:String end =
    tuple
        let a = 3
        and b = "four"
    end;
⇒ let t:Tuple a:Int b:String end = tuple let a=3 let b="four" end

```

Then the tuple components can be extracted by name via the dot notation:

```

✎ t.b;
⇒ "four": String

```

Note that a tuple may contain an arbitrary binding, including function definitions, recursive definitions, etc. An empty tuple type is the same as the Ok type, and the empty tuple is the same as ok.

Abstract tuples

An abstract type is obtained by a tuple type which contains a type and a set of constants and operations over that type.

```

✎ Let T = Tuple A::TYPE a:A f(x:A):Int end;
⇒ Let T::TYPE = Tuple A::TYPE a:A f(x:A):Int end

```

This abstract type can be implemented by providing a type, a value of that type, and an operation from that type to Int:

```

✎ let t1:T =
    tuple
        Let A::TYPE = Int
        let a:A = 0
        let f(x:A):Int = x+1
    end;
⇒ let t1:T =
    tuple
        Let A::TYPE=<Hidden>
        let a:A=<hidden>
        let f(x:A):Int=<fun>
    end

```

Abstract types and their values are not printed, in order to maintain the privacy implicit in the notion of type abstraction. Functions are not printed because they are compiled and their source code might not be easily available.

An abstract type can be implemented in many different ways; here is another implementation:

```

✎ let t2:T =
  tuple
    Let A::TYPE = Bool
    let a:A = false
    let f(x:A):Int = if x then 0 else 1 end
  end;
➡ let t2:T =
  tuple
    Let A::TYPE=<Hidden>
    let a:A=<hidden>
    let f(x:A):Int=<fun>
  end

```

What makes type T abstract? Abstraction is produced by the rule for extracting types and values from the tuples above: the identity of the type is never revealed, and values of that type are never shown:

```

✎ :t1.A;
➡ :t1.A :: TYPE
✎ t1.a;
➡ <hidden> : t1.A

```

Moreover, an abstract type does not match its own representation:

```

✎ t1.a + 1;
➡ Type Error: t1.A is not a subtype of Int

```

Although the representation type must remain unknown, it is still possible to perform useful computations:

```

✎ t1.f(t1.a);
➡ 1 : Int

```

But attempts to mix two different implementations of an abstract type produce errors:

```

✎ t1.f(t2.a);
➡ Type Error: t2.A is not a subtype of t1.A

```


This fails because it cannot be determined that t1 and t2 are the same implementation of T; in fact in this example they are really different implementations whose interaction would be meaningless.

☞ An abstract type can even be implemented as "itself":

```
☞ let t3:T =  
  tuple  
    Let A::TYPE = T  
    let a:A = t2  
    let f(x:A):Int = x.f(x.a)  
  end;
```

This is another instance of impredicativity where a value of an abstract type can contain its own type as a component.

☞ Define Booleans as an abstract type and write implementations in terms of integers, characters, or both. How can you emulate a conditional?

5.4. Option types

Option types model the usual disjoint union or variant record types; no unusual features are present here. An option type represents a choice between a finite set of signatures; objects having an option type come with a tag indicating the choice they represent. More precisely, an option type is an ordered named collection of signatures, where the names are distinct.

An object of an option type, or option, is a tuple consisting of a tag and a binding. An option of tag x and binding D has an option type T provided that T has a tag x with an associated signature S, and the binding D has signature S.

```
☞ Let T =  
  Option  
    a  
    b with x:Bool end  
    c with x,y:String end  
  end;  
☞ let aOption =  
  option a of T end;  
☞ let bOption =  
  option b of T with let x = true end;  
☞ let cOption =  
  option c of T with "xString" "yString" end;
```

Since the choices in an option type are ordered, there is an operator ordinal(o) returning the 0-based ordinal number of an option as determined by its tag. Note that the tag component of an option can never be modified.

Conversely, an option can be created from an ordinal number as `option ordinal(n) of T .. end`, where `n` is determined at run-time; this form is allowed only when all the branches of an option type have the same signature. ✂ This is not implemented.

The option tag can be tested by the `?` operator (see examples below).

The option contents can be extracted via the `!` operator. This may fail (at run-time) if the wrong tag is specified. If successful, the result is a tuple whose first component is the ordinal of the option.

```

✎ aOption?a;
➡ true : Bool
✎ bOption?a;
➡ false : Bool
✎ bOption!a;
➡ Exception: '!Error
✎ bOption!b;
➡ tuple 1 let x=true end : Tuple :Int x:Bool end
✎ bOption!b.x;
➡ true : Bool

```

A more structured way of inspecting options is through the case construct, which discriminates on the option tag and may bind an identifier to the option contents:

```

✎ case bOption
  when a then ""
  when b with arm then
    if arm.x then "true" else "false" end
  when c with arm then
    arm.x <> arm.y
  else "??
end;
➡ "true": String

```

A when clause may list several options, then the following are equivalent:

```

when x,y with z then a      when x with z then a
                             when y with z then a

```

where the two occurrences of `z` in the right-hand side must receive the same type.

If an option type starts directly with a with clause, it means that the components of that clause are common to all the branches. That is, the following types are equivalent:

<pre> Option with x:Bool end a with y:Bool end b with z:Bool end end;</pre>	<pre> Option a with x,y:Bool end b with x,z:Bool end end</pre>
---	--

However, values of these types behave slightly differently; in the former case, the common components can be extracted directly by the dot notation, without first coercing to a given label by !.

Option types will be discussed again in conjunction with subtypes.

5.5. Auto and Dynamic types

✂ These are not implemented, however dynamic types, below, are implemented and are a special case of auto types.

There are some situations in programming where it is necessary to delay typechecking until run-time. These situations can be captured by a new form of quantification: an infinite union of signatures indexed by types, whose use requires dynamic typechecking. This quantifier resembles both option types and abstract types. Like an option type it is a union with run-time discrimination, but the union is infinite. Like an abstract type it is a collection of types and related values, but the types can be dynamically inspected.

Automorphic values (auto values, for short) are values capable of describing themselves; they contain a type that is used to determine the rest of their shape. The related automorphic types (auto types) consist of a type variable X and a signature S depending on X ; they represent the union of all the signatures S when X varies over all types or, as we shall see, over all the subtypes of a given type.

```

✎ Let UniformPair =
    Auto A::TYPE with fst,snd:A end;
✎ let p:UniformPair =
    auto :Bool with false true end;
➡ let p:UniformPair = auto :Bool with false true end
```

🔒 Restriction: the type component of an auto value must be a closed type: it cannot contain any free type variables.

It is possible to discriminate on the type component of an automorphic object through a construct similar to case; this is named after the inspect construct in Simula67, since, as we shall see, it can be used to discriminate between the subtypes of a given type.

```

✎ inspect b
  when Bool with arm then arm.fst/arm.snd
  when Int with arm then (arm.fst*arm.snd) isnot 0
end;
➡ true: Bool

```

The types in the when clauses of inspect must all be closed types. The branch that is selected for execution (if any) is the first branch such that the type in the auto value is a subtype of the type in the branch. An else branch can be added at the end of the construct.

Automorphic types will be discussed again in conjunction with subtypes and with dynamic types.

5.5.1. Dynamic types

✂ Dynamics are currently unsound: the run-time type test is a no-op (this will only work in the future fully bootstrapped system).

Static typechecking cannot cover all situations. One problem is with giving a type to an eval function, or to a generic print function. A more common problem is handling in a type-sound way data that lives longer than any activation of the compiler [Atkinson Bailey Chisholm Cockshott Morrison 83]. These problems can be solved by the introduction of dynamic types, here realized by a built-in module `dynamic:Dynamic` (see the Appendix).

Objects of type `Dynamic_T` should be imagined as pairs of a type and an object of that type. In fact, `Dynamic_T` is defined as the type `Auto A::TYPE with a:A end`, and many of the relevant operations are defined in terms of operations on auto types. In particular, the type component of a `Dynamic_T` object must be a closed type, and it can be tested at run-time through `inspect`.

One can construct dynamic objects as follows:

```

✎ let d3:Dynamic_T = dynamic.new(:Int 3);
➡ let d3:Dynamic_T = auto :Int with 3 end

```

These objects can then be narrowed to a given (closed) type via the `dynamic.be` operation. If the given type matches the type contained in the dynamic, then the value contained in the dynamic is returned. Otherwise an exception is raised (narrowing is just a special case of `inspect`):

```

✎ dynamic.be(:Int d3);
➡ 3: Int
✎ dynamic.be(:Bool d3);
➡ Exception: dynamicError

```

The matching rules for narrowing and inspecting are the same as for static typechecking, except that the check happens at run-time.

Since an object of type `dynamic` is self-describing it can be saved to a file and then read back and narrowed, maybe in a separate programming session:

```

✎ let wr = writer.file("d3.dyn");
✎ dynamic.extern(wr d3);          (* Write d3 to file *)
✎ writer.close(wr);
...
✎ let rd = reader.file("d3.dyn");
✎ let d3 = dynamic.intern(rd);    (* Read d3 from file *)

```

✎ All values can be made into dynamics, including functions and dynamics.

☛ The operations `extern` and `intern` preserve sharing and circularities within a single dynamic object, but not across different objects.

☛ All dynamic values can be externed, except readers and writers; in general it is not meaningful to extern objects that are bound to input/output devices.

5.6. Recursive types

Recursive types are useful for defining lists, trees, and other inductive data structures. Since these are often parametric types, they will be discussed in the section on type operators.

Here we discuss a rather curious fact: recursive types allow one to reproduce what is normally called type-free programming within the framework of a typed language. Hence no "expressive power" (in some sense) is lost by using a typed language with recursive types. Again, this shows that a sufficiently sophisticated type system can remove many of the restrictions normally associated with static typing.

S-expressions are Lisp's fundamental data structures. These can be described as follows (for simplicity, we do not consider atoms):

```

✎ Let Rec SExp::TYPE =
    Option
      nil
      cons with car,cdr:SExp end
    end;
☛ Let Rec SExp::TYPE = Option nil cons with car,cdr:SExp end end

```

The usual Lisp primitives¹ can now be defined in terms of operations on options and tuples.

✎ Define such operations.

Another interesting recursive type is the "universal" type of "untyped" functions:

```

✎ Let Rec V::TYPE = All(:V)V;
☛ let Rec V::TYPE = All(:V)V

```

We can now define the universal untyped combinators `K` and `S`:

¹ I.e., "pure" Lisp primitives. To define `rplaca` see the next section.

```

✎ let K(x:V)(y:V):V = x;
➡ let K:V = <fun>
✎ let S(x:V)(y:V)(z:V):V = x(z)(y(z));
➡ let S:V = <fun>

```

✎ Note however that Quest is a call-by-value language; as an exercise, define a call-by-name version of the definitions above, and encode a non-strict conditional expression. (Hint: Let $\text{Rec } V::\text{TYPE} = \text{All}() \text{All}(:V)V.$)

✎ It is also interesting to notice that recursive types by themselves imply the presence of general recursive functions. As an exercise, define the fixpoint combinator on V (both the call-by-name and the call-by-value versions) without using `let rec`.

Two recursive types are equivalent when their infinite expansions (obtained by unfolding recursion) form equivalent infinite trees. Recursive type definitions describe finite graphs with type operators at the nodes, hence it is possible to test the condition above in finite time, without actually generating infinite structures [Courcelle 83].

5.7. Mutable types

Imperative programming is based on the notion of a mutable global store, together with constructs for sequencing the operations affecting the store.

✎ Mutability interacts very nicely with subtyping and polymorphism², showing that the functional-style approach suggested by type theory does not prevent the design of imperative languages.

5.7.1. Var types

Mutable store is supported by types of the form $\text{Var}(A)$, which is the type of mutable identifiers and data components of type A , and $\text{Out}(A)$, which is the type of write-only parameters of type A .

✎ These are not in fact first-class operators, in the sense that values of type $\text{Var}(A)$ or $\text{Out}(A)$ are not first-class values, and types like $\text{Var}(\text{Var}(A))$ cannot be produced. However, when discussing type rules it is often useful to think of `Var` and `Out` as ordinary operators; the above restrictions are for efficiency reasons only, and not for typing reasons.

An identifier can be declared mutable by prefixing it with the `var` keyword in a binding or signature; it then receives a `Var` type.

```


✎ let var a=3;
➡ let var a:Int = 3

```



² The problems encountered in ML [Gordon Milner Wadsworth 79, page 52] are avoided by the use of explicit polymorphism. However, we have to take some precautions with subtyping, as explained later.

The answer `let var a:Int = 3` above should be read as meaning that `a` has type `Var (Int)`.




When evaluating an expression of type `Var (A)` (or `Out (A)`), an automatic coercion is applied to obtain a value of type `A`.

```
 a;  
⇒ 3 : Int
```


The `:=` sign is used as the assignment operator; it requires a destination of type `Var (A)` and a source of type `A`, and returns `ok` after performing the side-effect:

```
 a:=5;  
⇒ ok : Ok  
 a+1;  
⇒ 6 : Int
```

✎ Functions cannot refer directly to global variables of `Var` type; these must be either passed as in and out parameters, or embedded in data structures to be accessed indirectly. This restriction facilitates the implementation of higher-order functions. Here is an example of two functions sharing a private variable that is accessible to them only (sometimes called an own variable):

```
 let t: Tuple f,g():Int end =  
  begin  
    let a = tuple let var x=3 end  
    tuple  
      let f():Int = begin a.x:=a.x+1 a.x end  
      let g():Int = begin a.x:=a.x+2 a.x end  
    end  
  end;  
⇒ let t:Tuple f():Int g():Int end =  
  tuple let f = <fun> let g = <fun> end  
  
 t.f();  
⇒ 4: Int  
  
 t.g();  
⇒ 6: Int
```

Data structure components can be made mutable by using the same `var` notation used for variables:

```
 let t = tuple let var a=0 end;  
⇒ let t = tuple let var a:Int = 0 end
```

```

✎ t.a := t.a+1;
⇒ ok : Ok

```

5.7.2. Sequencing and iteration

A syntactic construct called a block is used to execute expressions sequentially and then return a value. A block is just a binding D (hence it can declare local variables) with the restriction that its last component, which is the result value, must be a value expression. The type of a block is then the type of its last expression.

✎ Value-variables declared in a block $D:A$ may not appear free in A (as tuple variables from which an abstract type is extracted), since they would escape their scope.

The most explicit use of a block is in the `begin D end` construct, but blocks are implicit in most places where a value is expected, for example in the branches of a conditional.

Iteration is provided by a `loop .. exit .. end` construct, which evaluates its body (a binding) repeatedly until a syntactically enclosed `exit` is encountered. The `exit` construct has no type of its own (it can be thought of as an exception `raise exit as A end`, where A is inferred). A loop construct always returns `ok`, if it terminates, and has type `Ok`.

In addition, `while` and `for` constructs are provided as abbreviations for some loop-exit forms. (✎ Desugaring of `while` and `for` to be described.)

Here is an example of usage of blocks and iteration:

```

✎ let gcd(n,m:Int):Int =
  begin
    let var vn=n
    and var vm=m
    while vn isnot vm do
      if vn>vm then vn:=vn-vm end
      if vn<vm then vm:=vm-vn end
    end
    vn
  end;
⇒ let gcd(n,m:Int):Int = <fun>

✎ gcd(12 20);
⇒ 4 : Int

```

5.7.3. Out types

Formal parameters of functions may be declared of `Out` type by prefixing them with the `out` keyword. This makes them into write-only copy-out parameters.

When passing an argument corresponding to an `out` parameter, there must be an explicit coercion into an `Out` type. This is provided by the `@` keyword, which must be applied to a mutable

(Var or Out) entity: this is meant to be suggestive of passing the location of the entity instead of its value.

```
✎ let g(x:Int out y:Int):Ok = y:=x+1;  
✎ begin g(0 @t.a) t.a end;  
➡ 1 : Int
```

For convenience, it is also possible to convert a non-mutable value to an out parameter by creating a location for it on-the-fly, as in `g(0 let var y=0)`; this is also abbreviated as `g(0 var(0))`. In other words, an out entity can be obtained from a Var or Out entity by `@`, or from an ordinary value by `var`.

☞ This discussion was very operational; the deeper relations between Var and Out, and the type rules for passing Var and Out parameters, are explained in the section on subtyping.

✎ Polymorphism and Out parameters can be combined to define the following function:

```
✎ let assign(A::TYPE out a:A b:A):Ok = a:=b;  
✎ let var a=3;  
✎ assign(:Int @a 7);
```

that has the same effect as the assignment operator:

5.7.4. Array types

Array types `Array(A)` describe arrays with elements of type `A`, indexed by non-negative integers. Array types do not carry size information, since array values are dynamically allocated. Individual arrays are created with a given size, which then never changes.

Array values can be created with a given size and an initial value for all the elements, or from an explicit list of values. They can be indexed and updated via the usual square-bracket notation, and a extent operator is provided to extract their size.

```
✎ let a:Array(Int) = array of 0 1 2 3 4 5 end;  
✎ let b:Array(Bool) = array of(5 false);  
✎ b[0] := {a[0] is 0};
```

The array `a` has the 6 listed elements. The array `b` is defined to have 5 elements initialized to `false`.

We have used here listfix syntax, which is an enumeration of values enclosed in `of..end`, or a size-init pair enclosed in `of(..)`.

Although array is the basic listfix construct, additional listfix functions can be defined. They are interpreted according to the following abbreviations:

<code>f of .. end</code>	<code>for</code>	<code>f(array of .. end)</code>
<code>f of(n a)</code>	<code>for</code>	<code>f(array of(n a))</code>

Hence, all is needed to get a listfix function is to define an ordinary function with an array parameter:

```

✎ let sum(a:Array(Int)):Int =
    begin
        let var total=0
        for i = 0 upto extent(a)-1 do
            total := total+a[i]
        end
        total
    end;
➡ let sum(a:Array(Int)):Int = <fun>

✎ sum of 0 1 2 3 4 end;
➡ 10 : Int
✎ sum of(5 1);
➡ 5 : Int

```

☞ The complete set of operations on arrays is provided by the built-in module `arrayOp:ArrayOp`. It should be interesting to examine the polymorphic signatures of these operations.

5.8. Exception types

Exceptions, when used wisely, are an important structuring tool in embedded systems where anomalous conditions are out of the control of the programmer, or where their handling has to be deferred to clients.

Again it is comforting to notice that exceptions, being a control-flow mechanism, do not greatly affect the type system. In fact, exceptions can be propagated together with a typed entity, and in this sense they can be typechecked.

Many primitive operations produce exceptions when applied to certain arguments (e.g. division by zero). User-defined exceptions are also available in our language. Exceptions in Quest are treated differently than in most languages; there is an explicit notion of an exception value which can be created by the exception construct, and then bound to a variable or even passed to a function to be eventually raised.

```

✎ let exc1: Exception(Int) =
    exception integerException:Int end;
✎ let exc2: Exception(String) =
    exception stringException:String end;

```

The exception construct generates a new unique exception value whenever it is evaluated. The identifier in it is used only for printing exception messages; the type is the type of a value that can accompany the exception. Exceptions can be raised with a value of the appropriated type via the `raise` construct:

✎ raise exc1 with 3 end;
➡ Exception: integerException with 3:Int

Exceptions can be caught by the try construct that attempts evaluating an expression. If an exception is generated during the evaluation it will be matched against a set of exceptions, optionally binding an identifier to the exception value. If the exception is not among the ones considered by the try construct, and if there is no else branch, the exception is propagated further.

✎ try
 raise exc1 with 55 end
 when exc1 with x then x+5
 when exc2 then 1
 else 0
end;
➡ 60: Int

☛ Exceptions propagate along the dynamic call chain.

6. Operator kinds

Operators are functions from types to types that are evaluated at compile-time. There are also higher-order operators that map types or operators to other types or operators. All languages have built-in operators, such as function spaces and cartesian products, but very few languages allow new operators to be defined, or restrict them to first-order (types to types). Higher-order operators embody a surprising expressive power; they define one of the largest known classes of total functions [Girard 71], and every free algebra with total operations (booleans, integers, lists, trees, etc.) is uniformly representable in them [Böhm Berarducci 85] (see the example below for booleans). Because of this, we believe they will turn out to be very useful for parametrization and for carrying out compile-time computations.

An operator has the form:

$\text{Fun}(S)A$

where S is a signature introducing only type variables, and A is a type.

The kind of an operator has the form:

$\text{ALL}(S)K$

where K is a kind.

6.1. Type operators

The simplest operator is the identity operator, which takes a type and returns it; here it is defined in the normal and abbreviated ways.

```

✎ Let Id::ALL(A::TYPE)TYPE = Fun(A::TYPE) A;
✎ Let Id(A::TYPE)::TYPE = A;
➡ Let Id(A::TYPE)::TYPE = A

✎ :Id(Int);
➡ :Int :: TYPE
✎ let a:Id(Int) = 3;
➡ let a:Int = 3

```

The function space and cartesian product (infix) operators take two types and return a type:

```

✎ Let ->(A,B::TYPE)::TYPE = All(:A)B;
✎ Let #(A,B::TYPE)::TYPE = Tuple fst:A snd:B end;
✎ let f:{Int#Int}->Int = fun(p:Int#Int)p.fst+p.snd;

```

Note the omitted identifier in $All(:A)B$; this is to make typings such as $\{fun(x:A)B\}:A \rightarrow B$ legal for any identifier x , hence allowing f above to typecheck. Omitted identifiers and alpha-conversion were briefly discussed in the overview.

The optional operator is for data components which may or may not be present:

```

✎ Let Opt(A::TYPE)::TYPE =
    Option
    none
    some with some:A end
end;

```

Operator applications are evaluated with call-by-name, although this is not very important since all computations at the operator level terminate. (There are no recursive operators.)

Some interesting functional programs can be written at the operator level (DEF here introduces a kind):

```

✎ DEF BOOL = ALL(Then,Else::TYPE)TYPE;
✎ Let True::BOOL = Fun(Then,Else::TYPE) Then;
✎ Let False::BOOL = Fun(Then,Else::TYPE) Else;
✎ Let Cond(If::BOOL Then,Else::TYPE)::TYPE =
    If(Then Else);

```

📖 Convince yourself that Cond above is really a conditional at the type level.

✂ Rémy's pair-mapping example.

6.2. Recursive type operators

There are no recursive type operators, in order to guarantee that computations at the type level always terminate. However, there are recursive types which can achieve much of the same

effect. For example, one might expect to define parametric lists via a recursive operator as follows:

```
✎ (* Invalid recursive definition:
  Let Rec List(A::TYPE)::TYPE =
    Option
      nil
      cons with head:A tail:List(A) end
    end;
  *)
```

Instead, we have to define List as an ordinary operator that returns a recursive type:

✂ (We could use the syntax above as sugar for the syntax below.)

```
✎ Let List(A::TYPE)::TYPE =
  Rec(B)
    Option
      nil
      cons with head:A tail:B end
    end;
  ➡ Let List(A::TYPE)::TYPE =
    Rec(B::TYPE) Option nil cons with head:A tail:B end end
```

We have used the recursive type constructor $\text{Rec}(A)B$.

Here are the two basic list constructors:

```
✎ let nil(A::TYPE):List(A) =
  option nil of List(A) end
  and cons(A::TYPE)(head:A tail:List(A)):List(A) =
    option cons of List(A) with
      head tail
    end;
```

The other basic list operations are left as an exercise.

✂ Go look at the List library interface; it is an existential type containing an operator.

7. Power kinds

Most of the quantifier and operator structure described so far derives from Girard's system Fw [Girard 71]. The main new fundamental notion in Quest is the integration of subtyping with type quantification. This allows us to express more directly a wider range of programming situations and programming language features, as well as to introduce new ones.

In this section we define a subtyping relation between types, written $<:$, such that if x has type A and $A <: B$ then x has also type B . Hence, the intuition behind subtyping is ordinary set in-

clusion. Subtyping is defined for all type constructions, but normally holds only among types based on the same constructor. We do not have non-trivial subtype relations over basic types, although it is possible to add them when the machine representations of related types are compatible.

We can then talk of the collection of all subtypes of a given type B ; this forms a kind which we call $\text{POWER}(B)$, the power-kind of B . Then $A::\text{POWER}(B)$ and $A<:B$ have the same meaning; the former is taken as the actual primitive, while the latter is considered an abbreviation. Subtyping induces a subkind relation, written $<::$, on kinds; basically, $\text{POWER}(A) <:: \text{POWER}(B)$ whenever $A<:B$, and $\text{POWER}(A)<:: \text{TYPE}$ for any type A . Then we say that a signature S is a subsignature of a signature S' if S has the same number of components as S' with the same names and in the same position, and if the component types (or kinds) of S are subtypes (or subkinds) of the corresponding components in S' . Moreover, we say that S'' is an extended subsignature of S' if S'' has a prefix S which is a subsignature of S' . The formal subtyping rules for a simplified Quest language are in the Appendix.

Our notion of subtyping can emulate many characteristics of what is commonly understood as inheritance in object oriented languages. However, we must keep in mind that inheritance is a fuzzy word with no unequivocally established definition, while subtyping has a technical meaning, and that in any case subtyping is only one component of a full treatment of inheritance. Hence, we must try and keep the two notions distinct to avoid confusion.

The most familiar subtyping relation is between tuple types; this idea comes from object oriented programming where a class A is a subclass of another class B if the former has all the attributes (methods and instance variables) of the latter.

We interpret an object as a tuple, a method as a functional component of a tuple, a public instance variable as a modifiable value component of a tuple, a private instance variable as a modifiable own variable of the methods (not appearing in the tuple), a class as a function generating objects with given methods and instance variables, and finally a class signature as a tuple type. We do not discuss the interpretation of self here, which is in itself a very interesting and complex topic [Cook 87].

If A is a subclass of B , then A is said to inherit from B . In object oriented programming this usually (but not always) means that objects of A inherit the methods (functional fields) of B by sharing the method code, and also inherit the instance variables (data fields) of B by allocating space for them.

In our framework only signatures are in some sense inherited, not object components. Inheritance of methods can be achieved manually by code sharing. Since such sharing is not enforced by the language, we acquire flexibility: a class signature can be implemented by many classes, hence different instances of the same class signature can have different methods. This confers a dynamic aspect to method binding, while not requiring any run time search in the class hierarchy for method access.

7.1. Tuple subtypes

The subtyping rule for tuples is as follows. A tuple type A is a subtype of a tuple type B if the signature of A is an extended subsignature of the signature of B.

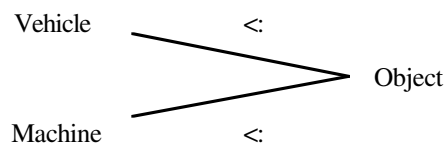
Remember that fields in a tuple or tuple type are ordered, hence tuple subtyping emulates single-inheritance hierarchies where the hierarchical structure always forms a tree (or a forest).

Simple tuples

Here is an example of a subtyping hierarchy on tuple types:

```
✎ Let Object =  
    Tuple age:Int end;  
✎ Let Vehicle =  
    Tuple age:Int speed:Int end;  
✎ Let Machine =  
    Tuple age:Int fuel:String end;
```

which produces the following subtyping diagram:



Here are a couple of actual objects:

```
✎ let myObject:Object =  
    tuple let age=3 end;  
✎ let myVehicle:Vehicle =  
    tuple let age=3 let speed=120 end;
```

Since `myVehicle:Vehicle` and `Vehicle<:Object`, we have that `myVehicle:Object` (this is called the subsumption rule). Hence functions expecting objects will accept vehicles:


```
✎ let age(object:Object):Int = object.age;  
✎ age(myVehicle);  
➡ 3 : Int
```


This example illustrates the flexibility inherent in subtypes. Note that inheritance (i.e. subtyping) is automatic and depends only on the structure of types; it does not have to be declared. Note also that the `age` function could have been defined before the `Vehicle` type, but still would work on vehicles as soon as these were defined.

Abstract tuples

Since tuples may be used to form abstract types, we immediately get a notion of abstract subtypes: that is, subtyping between abstract types. For example:

```

 Let Point =
    Tuple
        A::TYPE
        new(x,y:Int):A
        x,y(p:A):Int
    end;

 Let ColorPoint =
    Tuple
        A::TYPE
        new(x,y:Int):A
        x,y(p:A):Int
        paint(p:A c:Color):Ok
        color(p:A):Color
    end;

```


where a new color point starts with some default color and can then be painted in some other color. Here `ColorPoint <: Point`, hence any program working on points will also accept color points, by subsumption.


7.2. Option subtypes

The subtyping rule for option types is as follows: an option type A is a subtype of an option type B if B has all the tagged components of A (according to their names and positions) and possibly more, and the component signatures in A are extended subsignatures of the corresponding component signatures in B. Again, components in an option type are ordered.

For example, if we define the following enumeration types:

```


 Let Day =
    Option mon tue wed thu fri sat sun end;

 Let WeekDay =
    Option mon tue wed thu fri end;

```

then we have:

`WeekDay <: Day`

 It is important to notice that, unlike enumeration types in Pascal, here we can freely extend an existing enumeration with additional elements, while preserving type compatibility with the old enumeration. Similarly, after defining a tree-like data type using options, one can add new kinds of nodes without affecting programs using the old definition. Of course the old programs will not recognize the new options, but their code can be reused to deal with the old options.

 On the other hand, there is no automatic creation of values of enumeration types.

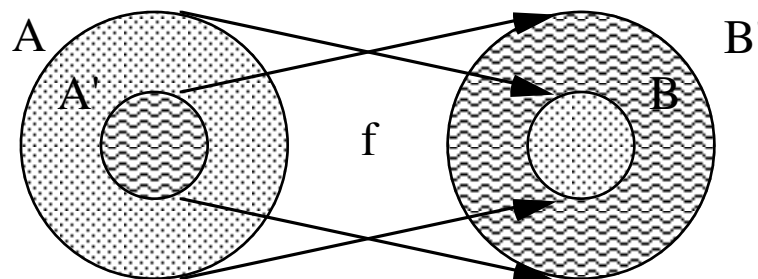
7.3. Function subtypes

In ordinary mathematics, a function from A to B can also be considered as a function from A' to B' if A' is a subset of A and B is a subset of B' . This is called contravariance (in the first argument) of the function space operator.

This also makes good programming sense: this rule is used in some form by all the object-oriented languages that have a sound type system. It asserts that a function working on objects of a given type will also work on objects of any subtype of that type, and that a function returning an object of a given type can be regarded as returning an object of any supertype of that type.

Hence the subtyping rule for function spaces asserts that $\text{All}(x:A)B <: \text{All}(x:A')B'$ if $A' <: A$ and (assuming $x:A'$) $B <: B'$.

By adopting this rule for functions, we take "ground" subtyping, at the data-structure level, and "lift" it to higher function spaces. Since object-oriented programming is based on ground subtyping, and



functional programming is based on higher-order functions, we can say that the rule above unifies functional and object-oriented programming, at least at the type level.

7.4. Bounded universals

Subtyping increases flexibility in typing, because of subsumption, but by the same mechanism can cause loss of type information. Consider the identity function on objects; when applied to a car, this returns an object:

```

let objectId(object:Object):Object = object;
objectId(myCar);
record age=3 end : Object

```

Here we have unfortunately forgotten that myCar was a Car, just by passing it through an identity function.

This problem can be solved by providing more type information:

```

let objectId(A<:Object)(object:A):A = object;
objectId(:Car)(myCar);
record age=3 speed=120 fuel="gas" end : Car

```

Note the signature $A <: \text{Object}$; `objectId` is now a polymorphic function, but not polymorphic in any type; it is polymorphic in the subtypes of `Object`. It takes a type which is any subtype of `Object`, then a value of that type, and returns a value of that type.

The signature $A <: \text{Object}$ is called a bounded (universal) quantifier; in fact it is not a new kind of signature because it can be interpreted as an abbreviation for $A::\text{POWER}(\text{Object})$.

Bounded universal quantifiers have the effect of integrating polymorphism with subtyping, and they provide more expressive power than either notion taken separately.

7.5. Bounded existentials

Bounded quantifiers can also be used in tuple types, where they are called bounded existential quantifiers. They provide a weaker form of type abstraction, called partial type abstraction; here for example we have an abstract type `A` which is not completely abstract: it is known to be an `Object`:

```
✎ Let U =  
    Tuple A<:Object a:A f(x:A):Int end;
```

It can be implemented as any subtype of `Object`; in this case it is implemented as a `Vehicle`.

```
✎ let u:U =  
    tuple  
        Let A<:Object = Vehicle  
        let a:A = myVehicle  
        let f(x:A):Int = x.speed  
    end;
```

✎ The interaction between type abstraction and inheritance is a delicate topic; in many languages inheritance violates type abstraction by providing access to implementation details of superclasses. Here we have mechanisms that smoothly integrate abstract types and subtyping: the partially abstract types above, and the abstract subtypes we have seen in a previous section.

7.6. Auto subtypes

An auto type is a subtype of another auto type if the respective components (with matching names and order) are in the subtype or subkind relation.

The first component of an auto type can be any subkind of `TYPE`; it is therefore possible to form a sum of a restricted class of types:

```
✎ Let AnyObject = Auto A<:Object with a:A end;  
✎ let aCar:AnyObject = auto :Car with myCar end;
```

Then, the `inspect` construct can be used to discriminate at run-time between the subtypes of a given type, in this case between the subtypes of `Object`. This operation is widely used in object-oriented languages such as `Simula67`, `Modula-3`, and `Oberon`.

7.7. Var, Out, and Array subtypes

The best way to explain the relations between Out and Var types is to take Out as a primitive operator, and consider Var(A) to be defined as a type conjunction [Reynolds --]:

$$\text{Var}(A) \quad \text{intended as} \quad \text{In}(A) \cap \text{Out}(A),$$

This means that Var(A) is the type of locations that have both type In(A) and type Out(A). Here In(A) is intended as the type of locations from which we can only read an A, and Out(A) as the type of locations to which we can only write an A. Hence Var(A) is the type of read-write locations.

Type conjunction has the following subtyping properties:

$$\begin{aligned} A \cap B &<: A & A \cap B &<: B \\ A <: B \wedge A <: C &\Rightarrow A <: B \cap C \end{aligned}$$

from which we may also derive (using reflexivity and transitivity of $<:$, and writing $A <:> B$ for $A <: B \wedge B <: A$):

$$\begin{aligned} A \cap A &<:> A \\ A \cap B &<:> B \cap A \\ A \cap (B \cap C) &<:> (A \cap B) \cap C \\ A <: C &\Rightarrow A \cap B <: C \\ A <: B \cap C &\Rightarrow A <: B \\ A <: B \wedge C <: D &\Rightarrow A \cap C <: B \cap D \end{aligned}$$

The conversions between A and In(A) are rather trivial; they are performed by let variable declarations ($A \rightarrow \text{In}(A)$) and by variable access. ($\text{In}(A) \rightarrow A$), with subtyping rule $A <: B \Rightarrow \text{In}(A) <: \text{In}(B)$. For simplicity, in Quest we identify the types A and In(A), and take their conversions as implicit. This amounts to assuming $\text{In}(A) <: A$ and $A <: \text{In}(A)$.

The Out types are much more interesting. First note that we assume neither $A <: \text{Out}(A)$ nor $\text{Out}(A) <: A$ (this would amount to $\text{In}(A) <: \text{Out}(A)$ and $\text{Out}(A) <: \text{In}(A)$). The fundamental subtyping rule for Out is suggested by the following reasoning: if every A is a B by subtyping, then a place where one can write a B is also a place where one can write an A. This means that Out is contravariant:

$$(1) \quad A <: B \Rightarrow \text{Out}(B) <: \text{Out}(A)$$

This has a nice consequence: out parameters appear in contravariant positions (on the "left" of function arrows), but as parameters they should behave covariantly since they are really function results. The contravariant rule for Out has the effect of neutralizing the contravariant rule for functions.

From the subtyping rules given so far we can easily infer:

- (2') $\text{Var}(A) <: A$
 (3') $\text{Var}(A) <: \text{Out}(A)$

with the more general forms:

- (2) $A <: B \Rightarrow \text{Var}(A) <: B$
 (3) $B <: A \Rightarrow \text{Var}(A) <: \text{Out}(B)$

and, putting the last two together:

- (4) $(A <: B) \wedge (B <: A) \Rightarrow \text{Var}(A) <: \text{Var}(B)$

Since $\text{Var}(A) <: A$ the conversion from the former to the latter is implicit and automatic; nothing special is needed to fetch the contents of a mutable location. This rule also says that any structure that can be updated can be regarded as a structure that cannot be updated, and passed to any function that will not update it.

Since $\text{Var}(A) <: \text{Out}(A)$, it becomes possible to pass a mutable location as an out parameter, but the explicit conversions `@` and `var` are required to prevent the automatic fetching of mutable locations described above³:

```
let var a = 3;
let f(x:Int out y:Int):Ok = y := x;
      (* i.e. f:Fun(x:Int y:Out(Int))Ok *)
f(3 @a);      (* a becomes 3 *)
f(3 var(0))    (* dummy location initially 0 becomes 3 *)
```

Now these conversions can be given a precise typing:

```
@: Out(A)->Out(A)
var: A->Var(A)
```

In the example, they have been used with types:

```
@: Var(Int)->Out(Int) :> Out(Int)->Out(Int)
var: Var(Int)->Out(Int) :> Int->Var(Int)
```

In-out var parameters could be used in addition to out parameters, but because of (4) they would be less flexible from a subtyping point of view. For simplicity, they are left out of the language⁴.

The following example shows that relaxing rule (4) to $A <: B \Rightarrow \text{Var}(A) <: \text{Var}(B)$ is semantically unsound:

³ These explicit conversions are required more for making programs easier to read than out of algorithmic necessity.

⁴ Note: the implicit conversion $\text{Var}(A) <: A$ is problematic for a natural implementation of in-out parameters as pointer-displacement pairs.

```

✎ let r =
    tuple
      let var a =
        tuple let b=3 end
      end;
✎ let f(r:Tuple var a:Tuple end end):Ok =
    r.a := tuple end;
✎ f(r);      (* Typechecks! *)
✎ r.a.b;     (* Crash! *)

```

Here the invocation of `f(r)` removes the `b` field from inside `r`, but `r` maintains its old type. Hence, a crash occurs when attempting to extract `r.a.b`. Note how everything typechecks under the weaker subtyping rule for `Var`.

Finally, the subtyping rule for arrays is give by considering them, for the purpose of subtyping, as functions from integers to mutable locations:

`Array(A)` regarded as `All(:Int) Tuple :Var(A) end`

Hence we obtain the rule:

$$(5) \quad (A <: B) \wedge (B <: A) \Rightarrow \text{Array}(A) <: \text{Array}(B)$$

which requires the `Array` operator to be both covariant and contravariant.

7.8. Recursive subtypes

Two recursive types are in subtype relation when their infinite expansions are in subtype relation. As for type equivalence, this condition can be tested in finite time. The formal version of this rule is non-trivial, and is described in appendix.

We can therefore produce hierarchies of recursively defined types. The following example also involves the subtyping rules for `Var` types, defined in the previous section.

```

✎ Let List(A::TYPE)::TYPE =
    Rec(List)
    Option
    nil
    cons with head:A tail:List end
end;

```

```

✎ Let VarList(A:TYPE)::TYPE =
    Rec(VarList)
    Option
    nil
    cons with var head:A var tail:VarList end
end;

```

Then we obtain, for example, $\text{VarList}(\text{Int}) <: \text{List}(\text{Int})$ by a non-trivial use of the $\text{Var}(A) <: A$ rule. Note that $\text{VarList} <: \text{List}$ does not make sense since subtyping is defined between types, not between operators.

8. Modularity

Most serious programming involves the construction of large programs, which have the property that no single person can understand or remember all of their details at the same time (even if they are written by a single person).

Large programs must be split into modules, not only to allow compilers to deal with them one piece at a time, but also to allow people to understand them one piece at a time. In fact it is a good idea to split even relatively small programs into modules. As a rule of thumb, a module should not exceed a few hundred lines of code, and can be as small as desired.

A lot of experience is required to understand where module boundaries should be located [Parnas 72]. In principle, any part of a program which could conceivably be reused should form a module. Any collection of routines which maintain an internal invariant that could be violated by careless use should also form a module. And almost every user-defined data type (or collection of closely related data types) should form a module, together with the relevant operations. Module boundaries should be located wherever there is some information that can or should be hidden; that is, information not needed by other modules, or information that can be misused by other modules.


8.1. Interfaces and modules

Module boundaries are called interfaces. Interfaces declare the types (or kinds) of identifiers supplied by modules; that is, they describe how modules may plug together to form systems.

Many interfaces provide one or more abstract data types with related operations; in this case they should define a consistent and complete set of operations on the hidden types. Other interfaces may just be a collection of related routines, or a collection of types to be used by other modules. Interfaces should be carefully documented, because they are the units of abstraction that one must understand in order to understand a larger system.

Both interfaces and modules may import other interfaces or modules; they all export a set of identifiers. A module satisfies (or implements) an interface if it provides all the identifiers required by the interface; a module may also define additional identifiers for its own internal purposes.

In Quest, interfaces and modules can be entered at the top level, although normally they will be found in separate files. Each interface, say A, can be implemented by many modules, say b and c. Each module specifies the interface it implements; this is written as b:A and c:A in the module headings:

 interface A	 module b:A
import ..	import ..
export	export
..	..
end;	end;



Both interfaces and modules may explicitly import other interfaces and modules in their headings (the interface implemented by a module is implicitly imported in the module). The following line imports: interfaces C, D, and E; module c implementing C; and modules d1 and d2 both implementing D.

```
import c:C d1,d2:D :E
✂ open c:C d1,d2:D :E
```

Imported modules are just tuples; values and types can be extracted with the usual "dot" notation. Imported interfaces are just tuple types. Since modules are tuples, they are first class: it is possible to pass modules around, store them in variables, and choose dynamically between different implementations of the same interface.

One of the main pragmatic goals of modules and interfaces is to provide separate compilation. When an interface is evaluated, a corresponding file is written, containing a compiled version of the interface. Similarly, when a module is evaluated, a file is written, containing a compiled version of the module. When interfaces and modules are evaluated, all the imported interfaces must have been compiled, but the imported modules do not have to be. The import dependencies of both modules and interfaces must form a directed acyclic graph; that is, mutually recursive imports are not allowed to guarantee that the linking process is deterministic.

Modules are linked by importing them at the top level⁵.

```
 import b:A;
 let b:A = ..
```

At this point all the interfaces and modules imported directly by b, A, and recursively by their imports, must have been compiled. The result is the definition at the top level of a tuple b, of type A, from which values and types can be extracted in the usual fashion.

A compiled module is like a function closure: nothing in it has been evaluated yet. At link time, the contents of compiled modules are evaluated in the context of their imports. In the link-

⁵ In view of dynamic linking, it might be nicer to introduce import as a new form of binding that would not be restricted to the top level.

ing process, a single copy of each module is evaluated (although it may be imported many times). Evaluation happens depth-first along import chains, and in the order in which modules appear in the import lists.

Version checking is performed during linking, to make sure that the compiled interfaces and modules are consistent.

8.2. Manifest types and kinds

When programming with modules, it is very convenient to define a type in an interface and then have other modules and interfaces refer to that definition. But so far we have shown no way of doing this: types can be defined in bindings (module bodies) where they are not accessible to other modules, and signatures (interfaces) can only contain specifications of abstract types⁶.

Hence a separate mechanism is provided to introduce type and kind definitions in signatures; such types and kinds are then called manifest. This should be seen just as a convenience: these types and kinds could be expanded at all their points of use, and everything would be the same. In particular, manifest entities appearing in signatures and bindings are (conceptually) removed and expanded before typechecking.

We augment our syntactic classes as follows:

- **Signatures:** they can now contain manifest kinds and types, introduced by the `DEF` and `Def` keywords respectively. The identifiers thus introduced can be mentioned later in the signature. These definitions are particularly useful in signatures forming tuple types (e.g. interfaces), so that these definitions can be later extracted from the tuple type.

`DEF U=K` is a manifest kind definition.

`Def X::K=A` is a manifest type definition. (Similarly for `Def Rec.`)

- **Kinds:** if `X` is a type identifier bound to a tuple type (e.g. an interface), and `U` is a kind variable introduced by `DEF` in that tuple type, then:

`X_U` is a kind.

- **Types:** if `A` denotes a tuple type, and `X` is a type variable declared by `Def` in that tuple type, then:

`A_X` is a type.

☞ The "dot" notation (`x.Y` and `x.y`) is used to extract types and values out of binding-like structures, while the "underscore" notation (`X_U` and `A_X`) is used to extract manifest kinds and types out of signature-like structures.









⁶ In fact we could write `X<:A` (a specification of a partially-abstract type) in an interface, thereby making `X` publicly available as (a subtype of) the known type `A`. However according to our signature matching rules, any implementation of such an interface should then "implement" `X`, probably just by duplicating `A`'s possibly complex definition.

8.3. Diamond import

In module systems that admit multiple implementations of the same interface, there must be a way of telling when implementations of an interface imported through different paths are the "same" implementation [MacQueen 84].

This is called the diamond import problem. A module *d* imports two modules *c* and *b* which both import a module *a*. Then the types flowing from *a* to *d* through two different import paths are made to interact in *d*. In other words, if *a*:*A* and *b*:*A*, then *a*.*T* must not match *b*.*T*, unless we know statically that *a* and *b* are the same value (the same implementation of the interface *A*).

In the Quest context (without parametric modules), this is solved by assuming a global name space of externally compiled modules, so that *a*.*T* matches *b*.*T* precisely when *a* and *b* are the same identifier.

 interface A export T::TYPE new(x:Int):T int(x:T):Int end;	 module a:A export Let T::TYPE = Int let new(x:Int):T = x and int(x:T):Int = x end;
 interface B import a:A export x:a.T end;	 module b:B import a:A export let x = a.new(0) end;
 interface C import a:A export f(x:a.T):Int end;	 module c:C import a:A export let f(x:a.T):Int = a.int(x)+1 end;
 interface D export z:Int end;	 module d:D import b:B c:C export let z = c.f(b.x) end;

Note that the application *c.f(b.x)* in module *d* typechecks because the *a* imported by *b* and the *a* imported by *c* are the "same" implementation of the interface *A*, since *a* is a global external name.

To illustrate the correspondence between interfaces and signatures, and between modules and bindings, we can rephrase the diamond import example as follows.

```

✎ Let A::TYPE =
  Tuple
    T::TYPE
    new(x:Int):T
    int(x:T):Int
  end;

```

```

✎ Let B::TYPE =
  Tuple
    x:a.T
  end;

```

```

✎ Let C::TYPE =
  Tuple
    f(x:a.T):Int
  end;

```

```

✎ Let D::TYPE =
  Tuple
    z:Int
  end;

```

```

✎ let a:A =
  tuple
    Let T::TYPE = Int
    let new(x:Int):T = x
    and int(x:T):Int = x
  end;

```

```

✎ let b:B =
  tuple
    let x = a.new(0)
  end;

```

```

✎ let c:C =
  tuple
    let f(x:a.T):Int = a.int(x)+1
  end;

```

```

✎ let d:D =
  tuple
    let z = c.f(b.x)
  end;

```

In this case, `c.f(b.x)` typechecks because the types of `c.f` and `b.x` both refer to the same variable `a` which is lexically in the scope of both `c` and `b`.

By convention, if an interface is dedicated to implementing a single type, or has a main type, that type is given the simple name "T", since this will always be qualified by the module name which will normally be expressive enough (e.g. `list.T`).

8.4. Soundness

In the Appendix, we provide a particular type violation mechanism via the built-in interface `Value`, which manipulates raw words of memory. But whatever the type violation mechanisms are, they need to be controlled somehow, lest they compromise the reliability of the entire language.

Cedar-Mesa and Modula-3 use the following idea. Unsound operations that may violate run-time system invariants are called *unsafe*. Unsafe operations can only be used in modules that are explicitly declared unsafe. If a module is declared safe, the compiler checks that (a) its body contains no unsafe operations, and (b) it imports no unsafe interfaces.

We adopt essentially the same scheme here, except we use the keyword `unsound`⁷. Unsound modules may advertise an unsound interface. However, unsound modules can also have ordinary interfaces. In the latter case, the programmer of the unsound module guarantees (i.e. proves or, more likely, trusts) that the module is not actually unsound, from an external point of view, although internally it uses unsound operations. This way, low-level facilities can be added to the system without requiring all the users of such facilities to declare their modules unsound just because they import them.

The only unsound operations in Quest are provided by built-in unsound interfaces such as `Value`. Unsound modules can import (implementations of) unsound interfaces and may have unsound or sound interfaces. Sound modules can import only (implementations of) sound interfaces. (These implementations may be unsound.) Sound interfaces are allowed to import (implementations of) unsound interfaces since soundness can be compromised only by operations, not by their types. Finally, the top level is implicitly sound.

9. Acknowledgments

Thanks for various discussions to: Roberto Amadio (rules for recursive types), Pierre-Louis Curien (typechecking algorithms), Peppe Longo (models of subtyping), and Benjamin Pierce (dynamic types).

⁷ Soundness is the property one proves to show that a type system does not permit type violations. The word *unsafe* has a slightly different meaning in the languages above, since some unsound operations are considered safe if they do not violate run-time invariants.

10. Appendix

10.1 Syntax

Id ranges over non-terminal identifiers; A and B range over syntactic expressions.

Id ::= A	the non-terminal identifier Id is defined to be A
Id	a non-terminal symbol
".."	a terminal symbol (" " is the empty input)
ide	an alphanumeric identifier token (A..Z, a..z, 0..9, with initial letter) or infix
infix	a symbolic identifier token (!@#\$\$%&* _+=- \`:<>/?)
char	a character token ('a', with escapes \n \b \t \f \\ \')
string	a string token ("abc", with escapes \n \b \t \f \\ \')
int	an integer token (2)
real	a real token (2.0)
A B	means A followed by B (binds strongest)
A B	means A or B
[A]	means (" " A)
{A}	means (" " A {A})
(A)	means A

Program ::=

{Phrase ";"}

Phrase ::=

[Interface | Module | Linkage | Binding |
"command" string | "load" string]

Interface ::=

["unsound"] "interface" ide ["import" Import]
"export" Signature "end"

Module ::=

["unsound"] "module" ide ":" ide ["import" Import]
"export" Binding "end"

Linkage ::=

("import" | "open") Import

Import ::=

{[ideList] ":" ide}

```

Kind ::=
  ide |
  "TYPE" |
  "POWER" "(" Type ")" |
  "ALL" "(" TypeSignature ")" Kind |
  ide "_" ide |
  "{" Kind "}"

Type ::=
  ide { "." ide | "!" ide } |
  "Ok" | "Bool" | "Char" | "String" | "Int" | "Real" |
  "Array" "(" Type ")" | "Exception" |
  "All" "(" Signature ")" Type |
  "Tuple" Signature "end" |
  "Option" OptionSignature "end" |
  "Fun" "(" TypeSignature ")" [HasKind] Type |
  "Rec" "(" ide ")" Type |
  Type "(" TypeBinding ")" |
  Type infix Type |
  Type "_" ide |
  "{" Type "}"

Value ::=
  ide |
  "ok" | "true" | "false" | char | string | integer | real |
  "if" Binding ["then" Binding] { "elseif" Binding ["then" Binding] }
    ["else" Binding] "end" |
  "begin" Binding "end" |
  "loop" Binding "end" | "exit" |
  "while" Binding "do" Binding "end" |
  "for" ide "=" Binding ( "upto" | "downto" ) Binding "do" Binding "end" |
  "fun" "(" Signature ")" [ ":" Type ] Value |
  Value "(" Binding ")" |
  Value (infix | "is" | "isnot" | "andif" | "orif" | "!=") Value |
  "tuple" Binding "end" |
  "option" (ide | $"ordinal" "(" Value ")")
    "of" Type ["with" Binding] "end" |
  Value ( "." | "?" | "!" ) ide |
  "case" Binding CaseBranches "end" |
  ("array" | Value) "of" (Binding "end" | "(" Binding ")" ) |
  Value "[" Value "]" [ ":" Value ] |

```

```

"exception" ide [":" Type] "end" |
"raise" Value ["with" Value] ["as" Type] "end" |
"try" Binding TryBranches "end" |
"{ " Value "}"

```

```

Signature ::=
{ TypeSignature |
  ["var" | "out"] IdeList (HasType|ValueFormals) | HasMutType }

```

```

TypeSignature ::=
{ "DEF" KindDecl |
  "Def" ["Rec"] TypeDecl |
  [IdeList] HasKind }

```

```

OptionSignature ::=
["with" Signature "end"] { IdeList ["with" Signature "end"]}

```

```

Binding ::=
{ "DEF" KindDecl |
  "Def" ["Rec"] TypeDecl |
  "Let" ["Rec"] TypeDecl |
  "let" ["rec"] ValueDecl |
  "::" Kind |
  ":" Type |
  "var" "(" Value ")" |
  "@" Value |
  Value }

```

```

TypeBinding ::=
{ Type }

```

```

KindDecl ::=
ide "=" Kind |
KindDecl "and" KindDecl

```

```

TypeDecl ::=
ide [HasKind | TypeFormals] "=" Type |
TypeDecl "and" TypeDecl

```

```

ValueDecl ::=
["var"] ide [HasType | ValueFormals] "=" Value |
ValueDecl "and" ValueDecl

```

```

TypeFormals ::=
{"(" TypeSignature ")" } HasKind

```

```

ValueFormals ::=
  {"(" Signature ")"} ":" Type

CaseBranches ::=
  {"when" IdeList ["with" ide] "then" Binding}
  ["else" Binding]

TryBranches ::=
  {"when" Binding ["with" ide] "then" Binding}
  ["else" Binding]

HasType ::=
  ":" Type

HasMutType ::=
  ":" Type | ":" "Var" "(" Type ")"

HasKind ::=
  "<:" Type | "::" Kind

IdeList ::=
  ide | ide ", " IdeList

```

Operators:

niladic:	Ok Bool Char String Int Real ok true false
prefix monadic:	not extent ordinal
infix:	is isnot andif orif
	$\wedge \vee + - * / \% < > <= >= <>$
	$++ -- ** // \wedge \wedge << >> <=> >=>$

Keywords:

ALL DEF POWER TYPE All Array Def Exception Fun Let Option Out Rec Tuple Var and array
 as begin case do downto else elsif end exception exit export for fun if import interface let loop
 module of open option out raise rec reraise then try tuple unsound upto var when while with ? ! :
 :: <: := = _ @

⌘ Other keywords:

Builtin builtin command load build

Notes:

The @ keyword can appear only in actual-parameter bindings. Bindings evaluated for a single result must end with a value component (modulo manifest declarations). Bindings in listfix constructs may begin with a type (the type of array elements) and must then contain only values. Recursive value bindings can contain only *constructors*, i.e. functions, tuples, etc. ⌘ Only functions

10.2 Type rules

A complete semantics of Quest would take many pages; we think it could be written using Structural Operational Semantics [Plotkin 81], also using the techniques developed in [Abadi Cardelli Pierce Plotkin 89] for dynamic types. See [Harper Milner Tofte 88] for a full formal language definition in this style.

This section contains the type rules for Miniature Quest, which is a language with scaled down notions of values, types, kinds, bindings, and signatures, but which presents the essential concepts.

Syntax

Signatures (S):	Types and operators (A,B,C; type ident. X,Y,Z):
\emptyset	X
S, X::K	All(S)A
S, x:A	Tuple(S)
	Fun(X::K)A A(B)
	Rec(X::TYPE)A
Bindings (D):	Values (a,b,c; value ident. x,y,z):
\emptyset	x
D, X::K=A	fun(S)a a(D)
D, x:A=a	tuple(D) bind S = a in b
Kinds (K, L, M):	rec(x:A)a
TYPE	
ALL(X::K)L	
POWER(A)	

Here we use the construct "bind S = a in b" which binds the components of the tuple "a" to the identifiers in "S" in the scope "b". In full Quest we use instead the notation "x.Y" and "x.y" to extract types and values out of a tuple denoted by an identifier "x". Then a program fragment "let x = a .. x.Y .. x.y..." corresponds to "bind ..Y::K..y:A.. = a in .. y .. Y...". The type rule for "bind" prevents the type identifiers in "S" from occurring in the type of the result. Similarly, in full Quest types like "x.Y" are not allowed to escape the scope of "x"; these restrictions are better understood by analogy with the "bind" construct.

Judgments

$\vdash S$ sig	S is a signature
$S \vdash K$ kind	K is a kind
$S \vdash A$ type	A is a type (same as $S \vdash A::\text{TYPE}$)

$S \vdash D::S'$	D has signature S'
$S \vdash A::K$	A has kind K
$S \vdash a:A$	a has type A
$S \vdash S'<::S''$	S' is a subsignature of S''
$S \vdash K<::L$	K is a subkind of L
$S \vdash A<:B$	A is a subtype of B (same as $S \vdash A::\text{POWER}(B)$)
$S \vdash S'<::>S''$	equivalent signatures
$S \vdash K<::>L$	equivalent kinds
$S \vdash A<::>B$	equivalent types

Notation

$S S'$ is the concatenation (iterated extension) of S with S' . Signatures and bindings are ordered sequences; however we freely use the notation $X \in \text{dom}(S)$ (type X is defined in S), $x \in \text{dom}(S)$ (value x is defined in S). Similarly for bindings.

$E\{X \leftarrow A\}$ and $E\{x \leftarrow a\}$ denote the substitution of the type variable X by the type A , or of the variable x by the value a , within an expression E of any sort. For a binding D , $E\{D\}$ is defined as follows: $E\{\emptyset\} = E$; $E\{D', X::K=A\} = E\{X \leftarrow A\}\{D'\}$; $E\{D', x:A=a\} = E\{x \leftarrow a\}\{D'\}$.

$E[E']$ indicates that E' is a given subexpression of expression E . Then $E[E'']$ denotes the substitution of (a particular occurrence of) E' by E'' in E . Here E , E' and E'' are expressions of any sort.

A type C is *contractive* in a (free) type variable X [MacQueen Plotkin Sethi 86], written $C \downarrow X$, if and only if C is either: a type variable different from X , a function or tuple type, an operator application whose reduced form is contractive in X , or a recursive type whose body is contractive in X . The body of a legal recursive type must also be contractive in the recursion variable. From a type whose recursion bodies are all contractive in their recursion variables, we can construct a well-formed regular (infinite) tree.

Equivalence

$\frac{S \vdash A::K}{S \vdash A<::>A}$	$\frac{S \vdash K \text{ kind}}{S \vdash K<::>K}$	$\frac{\vdash S S' \text{ sig}}{S \vdash S'<::>S'}$
$\frac{S \vdash A<::>A'}{S \vdash A'<::>A}$	$\frac{S \vdash K<::>K'}{S \vdash K'<::>K}$	$\frac{S \vdash S'<::>S''}{S \vdash S''<::>S'}$
$\frac{S \vdash A<::>A' \quad S \vdash A'<::>A''}{S \vdash A<::>A''}$	$\frac{S \vdash K<::>K' \quad S \vdash K'<::>K''}{S \vdash K<::>K''}$	$\frac{S \vdash S'<::>S'' \quad S \vdash S''<::>S'''}{S \vdash S'<::>S'''}$

Congruence

$\frac{S \vdash A[B]::K \quad S \vdash B<::>B'}{S \vdash A[B]<::>A[B']}$	$\frac{S \vdash A[K]::L \quad S \vdash K<::>K'}{S \vdash A[K]<::>A[K']}$	$\frac{S \vdash A[S']::L \quad S \vdash S'<::>S''}{S \vdash A[S']<::>A[S'']}$
--	--	---

$\frac{S \vdash K[A] \text{ kind} \quad S \vdash A <::> A'}{S \vdash K[A] <::> K[A']}$	$\frac{S \vdash K[L] \text{ kind} \quad S \vdash L <::> L'}{S \vdash K[L] <::> K[L']}$	$\frac{S \vdash K[S'] \text{ kind} \quad S \vdash S' <::> S''}{S \vdash K[S'] <::> K[S']}$
$\frac{\vdash S S'[A] \text{ sig} \quad S \vdash A <::> A'}{S \vdash S'[A] <::> S'[A']}$	$\frac{\vdash S S'[K] \text{ sig} \quad S \vdash K <::> K'}{S \vdash S'[K] <::> S'[K']}$	$\frac{\vdash S S'[S''] \text{ sig} \quad S \vdash S'' <::> S'''}{S \vdash S'[S''] <::> S'[S'']}$

Inclusion

$\frac{S \vdash A <::> A}{S \vdash A <:: A}$	$\frac{S \vdash K <::> L}{S \vdash K <:: K}$	$\frac{S \vdash S' <::> S''}{S \vdash S' <:: S''}$
--	--	--

Subsumption

$\frac{S \vdash a:A \quad S \vdash A <:: B}{S \vdash a : B}$	$\frac{S \vdash A::K \quad S \vdash K <:: L}{S \vdash A :: L}$	$\frac{S \vdash D::S' \quad S \vdash S' <:: S''}{S \vdash D :: S''}$
--	--	--

Conversion

$\frac{S, X::K \vdash B::L \quad S \vdash A::K}{S \vdash (\text{Fun}(X::K)B)(A) <::> B\{X \leftarrow A\}}$	$\frac{S \vdash \text{Rec}(X::\text{TYPE})A \text{ type}}{S \vdash \text{Rec}(X::\text{TYPE})A <::> A\{X \leftarrow \text{Rec}(X::\text{TYPE})A\}}$
$\frac{S \vdash A <::> C\{X \leftarrow A\} \quad S \vdash B <::> C\{X \leftarrow B\} \quad C \downarrow X}{S \vdash A <::> B}$	

Signatures

$\frac{}{\vdash \emptyset \text{ sig}}$	$\frac{S \vdash K \text{ kind} \quad X \notin \text{dom}(S)}{\vdash S, X::K \text{ sig}}$	$\frac{S \vdash A \text{ type} \quad x \notin \text{dom}(S)}{\vdash S, x:A \text{ sig}}$
---	---	--

Bindings

$\frac{\vdash S \text{ sig}}{S \vdash \emptyset :: \emptyset}$	$\frac{S \vdash D::S' \quad S \vdash A\{D\}::K\{D\}}{S \vdash D, X::K=A :: S', X::K}$	$\frac{S \vdash D::S' \quad S \vdash a\{D\}:A\{D\}}{S \vdash D, x:A=a :: S', x:A}$
--	---	--

Kinds

$\frac{\vdash S \text{ sig}}{S \vdash \text{TYPE} \text{ kind}}$	$\frac{S \vdash K \text{ kind} \quad S, X::K \vdash L \text{ kind}}{S \vdash \text{ALL}(X::K)L \text{ kind}}$	$\frac{S \vdash A \text{ type}}{S \vdash \text{POWER}(A) \text{ kind}}$
--	---	---

Types and Operators

$\frac{\vdash S, X::K, S' \text{ sig}}{S, X::K, S' \vdash X :: K}$	$\frac{S S' \vdash A \text{ type}}{S \vdash \text{All}(S')A \text{ type}}$	$\frac{\vdash S S' \text{ sig}}{S \vdash \text{Tuple}(S') \text{ type}}$
$\frac{S, X::K \vdash B::L}{S \vdash \text{Fun}(X::K)B :: \text{ALL}(X::K)L}$	$\frac{S \vdash B::\text{ALL}(X::K)L \quad S \vdash A::K}{S \vdash B(A) :: L\{X \leftarrow A\}}$	$\frac{S, X::\text{TYPE} \vdash A \text{ type} \quad A \downarrow X}{S \vdash \text{Rec}(X::\text{TYPE})A \text{ type}}$

Values

$\frac{\vdash S, x:A, S' \text{ sig}}{S, x:A, S' \vdash x : A}$	$\frac{S S' \vdash a:A}{S \vdash \text{fun}(S')a : \text{All}(S')A}$	$\frac{S \vdash a:\text{All}(S')A \quad S \vdash D \vdash S'}{S \vdash a(D) : A\{D\}}$
$\frac{S \vdash D \vdash S'}{S \vdash \text{tuple}(D) : \text{Tuple}(S')}$	$\frac{S \vdash a:\text{Tuple}(S') \quad S \vdash B \text{ type} \quad S S' \vdash b:B}{S \vdash \text{bind } S' = a \text{ in } b : B}$	$\frac{S, x:A \vdash a : A}{S \vdash \text{rec}(x:A)a : A}$

SubSignatures

$\frac{S \vdash S' <: S'' \quad S S' \vdash K <: L}{S \vdash S', X::K <: S'', X::L}$	$\frac{S \vdash S' <: S'' \quad S S' \vdash A <: B}{S \vdash S', x:A <: S'', x:B}$
--	--

SubKinds

$\frac{S \vdash K' <: K \quad S, X::K' \vdash L <: L'}{S \vdash \text{ALL}(X::K)L <: \text{ALL}(X::K')L'}$	$\frac{S \vdash A \text{ type}}{S \vdash \text{POWER}(A) <: \text{TYPE}}$	$\frac{S \vdash A <: B}{S \vdash \text{POWER}(A) <: \text{POWER}(B)}$
--	---	---

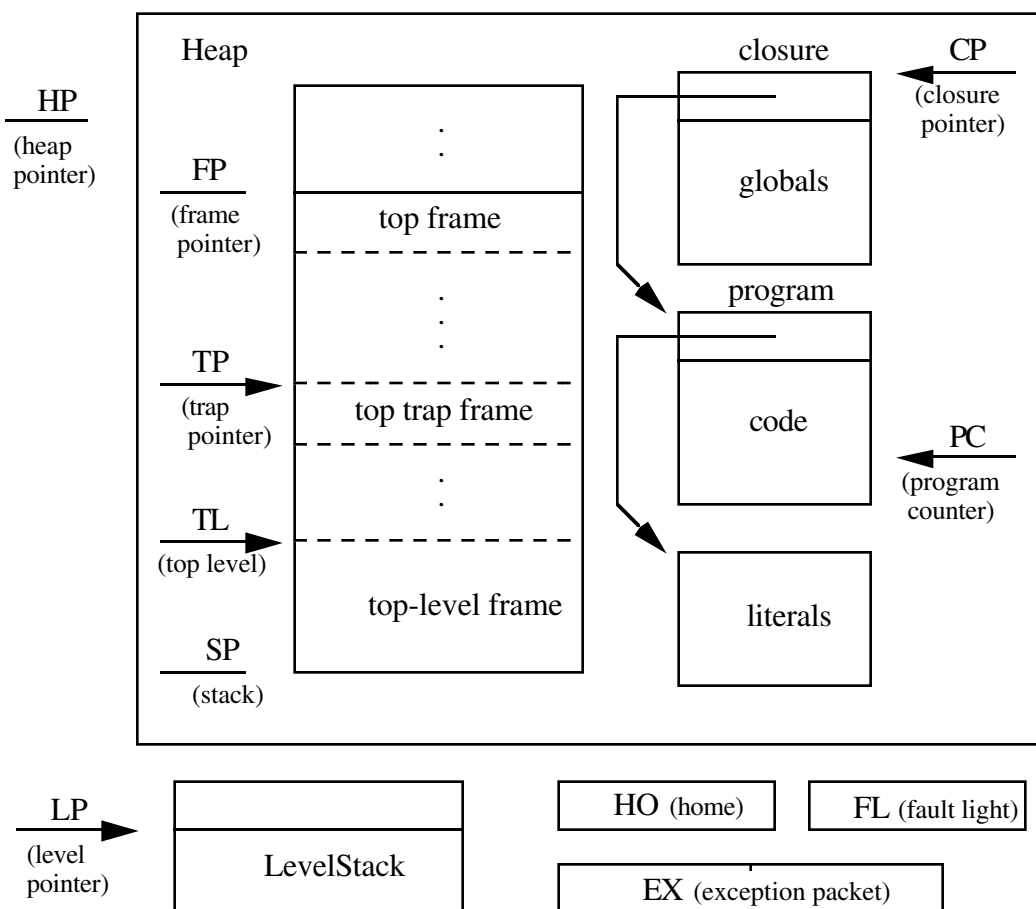
SubTypes

$\frac{S \vdash S' <: S'' \quad S, S'' \vdash A' <: A''}{S \vdash \text{All}(S')A' <: \text{All}(S'')A''}$	$\frac{\vdash S S'S'' \text{ sig} \quad S \vdash S' <: S''}{S \vdash \text{Tuple}(S'S'') <: \text{Tuple}(S'')}$
$\frac{S \vdash \text{Rec}(X::\text{TYPE})A \text{ type} \quad S \vdash \text{Rec}(Y::\text{TYPE})B \text{ type} \quad S, Y::\text{TYPE}, X<:Y \vdash A <: B}{S \vdash \text{Rec}(X::\text{TYPE})A <: \text{Rec}(Y::\text{TYPE})B}$	

10.3 Quest Machine

Quest programs are compiled into an intermediate code format that can then be use as a start-
ing point for code generation. This code can also be run directly on an abstract machine, called
the Quest Machine, which is described in this section.

In the current system, the intermediate code is assembled into *bytecode* (a simple lineariza-
tion of the intermediate code), which is then interpreted by a Modula3 program emulating the
Quest Machine. Other back-end implementations might assemble the intermediate code into, e.g.,
native VAX code and run it on a VAX architecture.



M (machine state) = $\langle SP, TL, TP, FP, CP, PC, EX, HO, FL \rangle$

G (global state) = $\langle \text{Heap}, HP, \text{LevelStack}, LP, M \rangle$

Figure 1: The Quest Machine

The global state of the Quest Machine is described in Figure 1. It consists of a *heap*, a gen-
eral purpose register called *home* (*HO*), an *exception packet* (*EX*) register used during exception
handling (where *EX1* is the exception name and *EX2* the exception value), a *fault light* (*FL*) indi-
cating normal or exceptional termination (initially off), and a *level stack* (*LP*), (initially empty).

The level stack can store machine states (described below) to keep track of recursive invocations of the machine. Typically the Quest system, which is a Quest program running at *level 0*, compiles another Quest program and recursively starts a machine at *level 1* to execute it. When the level 1 machine stops, the control returns to the machine at *level 0*, that is to the Quest system. Sometime a Quest system is run on top of another Quest system for debugging; then there may be three active levels. A main feature of this scheme is that exceptions do not propagate through level boundaries.

The heap has a *heap pointer (HP)* to the unallocated area. The allocated area contains a current *stack*, pointed by a *stack pointer (SP)*, and a current *closure*, pointed by a *closure pointer (CP)*.

The closure contains a record of all the global variables of the current program, and a pointer to the *program*. The program is a piece of code, with the *program counter (PC)* indicating the current instruction, plus a pointer to a record of all the literals of the program. Literals are typically strings, or other programs arising from statically nested functions. Program code is relocatable, and contains no pointers into the heap; all such pointers are collected as literals. A closure is a completely self-contained entity; it contains all it needs to execute, with the possible exception of I/O handles gone stale.

The stack has a large bottom frame, pointed by a *top-level pointer (TL)*; this is where the values declared at the top level are kept. Above the top level are the frames resulting from function activations; the topmost (current) frame is indicated by the *frame pointer (FP)*; initially FP=TL. Exception handlers are threaded through the active frames; the topmost, if any, is indicated by the *trap pointer (TP)*; initially TP=SP. The stack is relocatable.

Finally, a complete *machine state*, is determined by SP, TL, FP, CP, PC, EX, HO and FL.

The structure of a frame is described in Figure 2. From bottom to top there is a (currently unused) frame descriptor, space for function results (only single results are implemented), space for arguments, space for temporaries, and two locations to save CP and PC on function calls. Addressing conventions are indicated in the figure.

The calling conventions require the caller to assemble the arguments into the (future) frame above the current FP; the caller knows the number of arguments and results of the callee. The caller then saves CP and PC, and the callee is invoked to initialize a proper frame around the arguments. The callee knows how many temporaries it requires, and raises FP accordingly. Eventually the callee stores its results in the result space, lowers FP of the known amount, and restores the previous CP and PC. The caller gets control again and fetches the results of the callee. (Note: FP's are not threaded. Stack unwinding, e.g. for debugging, can be done via the CPs stored in each frame, since the first instruction of each program has the frame size in it.) (Note: in presence of concurrent garbage collection, a *shadow FP* must be maintained to prevent the collector from reclaiming data temporarily stored above the real FP.)

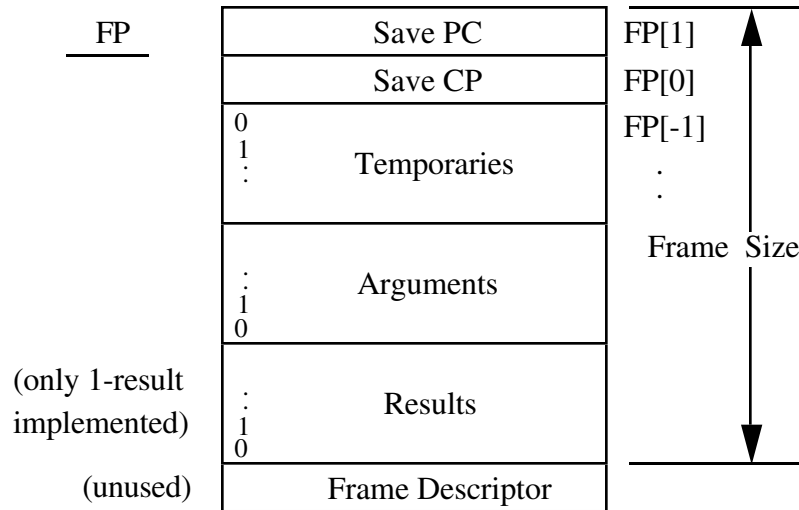


Figure 2: Frame structure

Addressing is described in the following table.

Immediate(imm)	referring directly to imm, where imm is either ok, a boolean, a character, an integer or a real. Read only.
Frame(n)	referring to the contents of the location FP+n (n pointees).
FrameLocative(n)	referring to the contents of the m-th pointee off the address found at FP+n, where m is found at FP+n+pointersize. Used to pass parameters by var.
Literal(n)	referring to the contents of the n-th literal location (n≥0 pointees), obtained via CP.
Global(n)	referring to the contents of the n-th global location (n≥0 pointees), obtained via CP.
Indexed(A,n)	referring to the contents of the n-th pointee off the address described by A, where A is another addressing.
StackDispl(n)	Referring directly to (FP-SP)+n (n≤0 pointees). Used to obtain the relative stack location of a frame location, in conjunction with var parameters. Read only.
Special(s)	referring, depending on s, to the contents of either HO, SP, EX1 (exception name, read only), or EX2 (exception value, read only).

Finally, here is an informal description of the instruction set.

NoOp() do nothing.

Move(A₁, A₂) fetch a value from A₁ and store it in A₂.

Arguments(resultsNo, (A₁, ..., A_n)) move the values from A₁..A_n in the "future" frame above FP. Also clear the result slots.

Apply(A) call the closure at A. That is, save the current CP and PC in the reserved slots of the current frame, set CP to the new closure, and set PC to the beginning of the new program.

Frame(frameSize,tempSize) first instruction of each program frame initialized given size by raising FP and clearing the temporary slots (assume argument and result slots are ready).

Return(frameSize) lower FP according to frameSize; the results of the returning program remain in the "past" frame above the FP. Restore the CP and PC of the calling closure.

Results((A₁,...,A_n)) move the results of the "past" frame into locations A₁..A_n of the current frame.

Closure(A₁, (A₂,...,A_n), A_{n+1}) allocate a closure with program from A₁ (usually a literal) and globals from A₂..A_n. Store the result at A_{n+1}.

DumClosure(A₁, globalsSize, A₂) allocate a closure with program from A₁ and globalsSize null globals. Store the result at A₂.

RecClosure(A₁, (A₂,...,A_n)) fill the globals of the closure found at A₁ with the values at A₂..A_n. This is intended to match a previous DumClosure.

JumpAlways(target) set PC to PC+target (relative to the address of target).

Jump(condition, A₁, A₂, target) if condition is true of the values at A₁, A₂, set PC to PC+target (relative to the address of target). Conditions are: True, False, Eq, NotEq, Less, LessEq, More, MoreEq.

Case(A, (target₁,...,target_n)) get the tuple value at A; let its first field contain the number $i \geq 0$. Set PC to PC+target_i (relative to the address of target_i).

CaseFault() store a case-fault exception in EX and do Raise+Unwind.

CaseCheck(A, i) check that the first field of the tuple value at A is i; otherwise store a case check exception in EX and do Raise+Unwind.

Trap(n, target) store a (relocation independent) trap frame at location n in the current frame, with exception handler to be later found at target. That is, let PC₀(CP) be the address of the first instruction of the program at CP, and tf be FP+n; then: tf[0]^ := TP-SP; tf[-1]^ := CP; tf[-2]^ := PC+target-PC₀(CP); tf[-3] := FP-SP; TP := tf.

Untrap forget the topmost trap frame: TP := TP^+SP.

Raise(A₁, A₂) begin raising the exception whose unique identifier is at A₁ (a string) and whose exception value is at A₂. That is, move A₁ to EX1 and A₂ to EX2. An Unwind should then follow.

Unwind() perform one step of stack unwinding. That is; if TP=Stack, set FL on (signalling a fault) and do a Stop. Otherwise restore the machine to the state recorded in the last trap frame: tf := TP; TP := tf[0]^+SP; CP := tf[-1]^; PC := tf[-2]^+PC₀(CP); FP := tf[-3]^+SP. Exception matching and further unwinding must be programmed in, at the beginning of every exception handler.

Crash() store a crash exception in EX and do Raise+Unwind.

Start(A_1, A_2) fetch a data structure from A_1 representing a machine state M , push the current machine state on the level stack, and start the new machine with M . A_2 is later fetched by the Stop instruction of M .

Stop() if the level stack is empty then set FL off and stop the machine. Otherwise create a machine state M for the current machine, pop a machine state from the level stack and make it current, and store M in the A_2 address from the suspended Start instruction.

Data(op, (A_1, \dots, A_n), A_{n+1}) perform the data operation op with arguments from $A_1..A_n$ and result in A_{n+1} . There is an extensive list of such operations for all the built-in data types and modules.

10.4 Library interfaces

Here is a list of supplied modules; their interface specification follows. Most of these modules are pre-linked at the top level, but must be explicitly imported by any module which uses them.

☛ The ones that are not implicitly imported can be imported explicitly, and will be found.

arrayOp: ArrayOp	(* Array operations *)
ascii: Ascii	(* Ascii conversions *)
conv: Conv	(* String conversions *)
dynamic: Dynamic	(* Dynamically typed values *)
int: IntOp	(* Integer number operations *)
list: List	(* Polymorphic lists *)
reader: Reader	(* Input operations *)
real: RealOp	(* Real number operations *)
trig: Trig	(* Trigonometry, to be defined *)
string: StringOp	(* String operations *)
value: Value	(* Arbitrary operations. Unsound! *)
writer: Writer	(* Output operations *)

⌘ Always check these interfaces in the current system, to see what the real truth is. These however should be pretty close.

interface ArrayOp

export

error:Exception(Ok)

(* Raised when an operation cannot be carried out. *)

new(A::TYPE size:Int init:A):Array(A)

(* Create a new array of given size, all initialized to init. *)

size(A::TYPE arr:Array(A)):Int

(* Return the size of an array, same as "extent(arr)". *)

get(A::TYPE arr:Array(A) index:Int):A

(* Extract an array element, same as "arr[index]". *)

set(A::TYPE arr:Array(A) index:Int item:A):Ok

(* Update an array element, same as "arr[index] := item". *)

end;

interface Ascii

export

error:Exception(Ok)

(* Raised when an operation cannot be carried out. *)

char(n:Int):Char

```

        (* Return the character of ascii encoding n. Raise error if n<0
           or n>255. *)
    val(c:Char):Int
        (* Return the ascii encoding of character c. *)
end;

interface Conv
export
    okay():String
        (* Return the string "ok". *)
    bool(b:Bool):String
        (* Return "true" if b is true, "false" otherwise. *)
    int(n:Int):String
        (* Return a string representation of the integer n (preceded
           by '~' if negative). *)
    real(r:Real):String
        (* Return a string representation of the real r (preceded
           by '~' if negative). *)
    char(c:Char):String
        (* Return a string containing the character c in single quotes,
           with backslash encoding if necessary. *)
    string(s:String):String
        (* Return a string containing the string s in double quotes,
           with backslash encoding wherever necessary. *)
end;

interface Dynamic
import reader:Reader writer:Writer
export
    T::TYPE
        (* A pair of an arbitrary object and its type. *)
    error:Exception(Ok)
        (* Raised when an operation cannot be carried out. *)
    new(A::TYPE a:A):T
        (* Package an object of any type into an object of type T. *)
    be(A::TYPE d:T):A
        (* If the object d was generated from an object a of type A,
           then return a, otherwise raise error. *)
    copy(d:T):T
        (* Make a complete copy of a dynamic object. *)
    intern(rd:reader.T):T

```

```

    (* Read a dynamic object from a reader. *)
extern(wr:writer.T d:T):Ok
    (* Write a representation of a dynamic object to a writer. *)
end;

interface IntOp
export
    error:Exception(Ok)
        (* Raised when an operation cannot be carried out. *)
    minInt,maxInt:Int
        (* The most negative and most positive representable integers. *)
    abs(r:Real):Real
        (* Absolute value. *)
    min,max(r:Real):Real
        (* Min and max. *)
end;

interface List
export
    T:: ALL(::TYPE)::TYPE
        (* If list:List is an implementation of this interface,
           then list.T(A) is a list of items of type A. *)
    error:Exception(Ok)
        (* Raised when an operation cannot be carried out. *)
    nil(A::TYPE):T(A)
        (* The (polymorphic) empty list. *)
    cons(A::TYPE :A :T(A)):T(A)
        (* List construction. *)
    null(A::TYPE :T(A)):Bool
        (* Whether a list is empty. *)
    head(A::TYPE :T(A)):A
        (* Return the head of a list. *)
    tail(A::TYPE :T(A)):T(A)
        (* Return the tail of a list. *)
    length(A::TYPE :T(A)):Int
        (* Return the length of a list. *)
    enum(A::TYPE :Array(A)):T(A)
        (* "list.enum of ... end" returns a list of elements "...". *)
end;

```

```

✂ interface Nester
    (Basic prettyprinter. Works, see interface)

```

```

interface Reader
export
  T::TYPE
    (* A reader is a source of characters. *)
  error:Exception(Ok)
    (* Raised when an operation cannot be carried out. *)
  input:T
    (* The standard input reader. *)
  file(name:String):T
    (* Make a reader out of a file, given its file name. *)
  more(reader:T):Bool
    (* Test for end of character stream. May block. *)
  ready(reader:T):Int
    (* Counts the number of characters that can be read without
       blocking; the end-of-stream marker counts as 1.
       Never blocks. *)
  getChar(reader:T):Char
    (* Read a character from a reader. May block. *)
  getString(reader:T size:Int):String
    (* Read a string of given size from a reader. May block. *)
  getSubString(reader:T string:String start,size:Int):Ok
    (* Read a string of given size from a reader and store it in
       another string at a given position. May block. *)
  close(reader:T):Ok
    (* Close a reader; operations on it will now fail. *)
end;

```

```

interface RealOp
export
  error:Exception(Ok)
    (* Raised when an operation cannot be carried out. *)
  minReal,maxReal:Real
    (* The most negative and most positive representable reals. *)
  negEpsilon,posEpsilon:Real
    (* The most positive representable negative real, and
       the most negative representable positive real. *)
  e:Real
    (* e. *)
  int(n:Int):Real
    (* Convert an integer to a real. *)

```

```

floor,round(r:Real):Int
    (* Convert a real to an integer. *)
plus,diff,mul,div,exp(r:Real):Real
    (* Basic operations, A.k.a. ++ -- ** // ^^. *)
smaller,greater,smallerEq,greaterEq(r,s:Real):Bool
    (* Basic comparisons, A.k.a. << >> <=> >=>. *)
abs,log(r:Real):Real
    (* Abs and log. *)
min,max(r:Real):Real
    (* Min and max. *)
end;

interface StringOp
export
    error:Exception(Ok)
        (* Raised when an operation cannot be carried out. *)
    new(size:Int init:Char):String
        (* Create a new string of give size, all initialized to init. *)
    isEmpty(string:String):Bool
        (* Test whether a string is empty. *)
    length(string:String):Int
        (* Return the length of a string. *)
    getChar(string:String index:Int):Char
        (* Extract a character from a string. *)
    setChar(string:String index:Int char:Char):Ok
        (* Replace a character of a string. *)
    getSub(source:String start,size:Int):String
        (* Extract a substring from a string. *)
    setSub(dest:String destStart:Int
        source:String sourceStart,sourceSize:Int):Ok
        (* Replace a substring of a string. If source and dest are the
            same, copies characters in the appropriate direction. *)
    cat(string1,string2:String):String
        (* Concatenate two strings (same as "<>"). *)
    catSub(string1:String start1,size1:Int
        string2:String start2,size2:Int):String
        (* Concatenate two substrings. *)
    conc(a:Array(String)):String
        (* "string.conc of s1..sn end" is the concatenation of s1..sn. *)
    equal(string1,string2:String):Bool
        (* True if two strings have the same size and contents. *)

```

```

equalSub(string1:String start1,size1:Int
         string2:String start2,size2:Int):Bool
    (* True if two substrings have the same size and contents. *)
precedes(string1,string2:String):Bool
    (* True if two strings are equal or in Ascii
       lexicographic order. *)
precedesSub(string1:String start1,size1:Int
            string2:String start2,size2:Int):Bool
    (* True if two substrings are equal or in Ascii lexicographic
       order. *)
end;

unsound interface Value
export
  T::TYPE
    (* The type of an arbitrary value. *)
  error:Exception(Ok)
    (* Raised when an operation cannot be carried out. *)
  length:Int
    (* Implementation-dependent size of a value in implementation-
       dependent units. *)
  new(A::TYPE a:A):T
    (* Convert any value to a T. *)
  be(A::TYPE v:T):A
    (* Convert a T to a value of a given type. Unsound! *)
  fetch(run:T index:Int):T
    (* Fetch the value at location run+index in memory; index is in
       implementation-dependent units. *)
  store(run:T index:Int val:T):Ok
    (* Store a value at location run+index in memory; displ is in
       implementation-dependent units. Unsound! *)
end;

interface Writer
export
  T::TYPE
    (* A writer is a sink of characters. *)
  error:Exception(Ok)
    (* Raised when an operation cannot be carried out. *)
  output:T
    (* The standard output writer. *)

```

```
file(name:String):T
    (* Make a writer out of a file, given its file name. *)
flush(writer:T):Ok
    (* Flush any buffered characters to their final destination. *)
putChar(writer:T char:Char):Ok
    (* Write a character to a writer. *)
putString(writer:T string:String):Ok
    (* Write a string to a writer. *)
putSubString(writer:T string:String start,size:Int):Ok
    (* Write a substring of a given string to a writer. *)
close(writer:T):Ok
    (* Close a writer; operations on it will now fail. *)
end;
```

References

- [Abadi Cardelli Pierce Plotkin 89] M.Abadi, L.Cardelli, B. Pierce, G.D.Plotkin: Dynamic Typing in a Statically Typed Language, Proc. POPL 1989.
- [Atkinson Bailey Chisholm Cockshott Morrison 83] M.P.Atkinson, P.J.Bailey, K.J.Chisholm, W.P.Cockshott, R.Morrison: An approach to persistent programming, Computer Journal 26(4), November 1983.
- [Barendregt 85] H.P.Barendregt: The lambda-calculus, its syntax and semantics, North-Holland 1985.
- [Böhm Berarducci 85] C.Böhm, A.Berarducci: Automatic synthesis of typed λ -programs on term algebras, Theoretical Computer Science, 39, pp. 135-154, 1985.
- [Burstall Lampson 84] R.M.Burstall, B.Lampson: A kernel language for abstract data types and modules, in Semantics of Data Types, Lecture Notes in Computer Science 173, Springer-Verlag, 1984.
- [Cardelli Wegner 85] L.Cardelli, P.Wegner: On understanding types, data abstraction and polymorphism, Computing Surveys, Vol 17 n. 4, pp 471-522, December 1985.
- [Cardelli 86] L.Cardelli: Amber, Combinators and Functional Programming Languages, Proc. of the 13th Summer School of the LITP, Le Val D'Ajol, Vosges (France), May 1985. Lecture Notes in Computer Science n. 242, Springer-Verlag, 1986.
- [Cardelli 88] L.Cardelli: Types for Data-Oriented Languages, Proceedings of the First Conference on Extending Database Technology, Venice, Italy, March 14-18, 1988.
- [Cardelli Donahue Glassman Jordan Kalsow Nelson 88] L.Cardelli, J.Donahue, M.Jordan, B.Kalsow, G.Nelson: Modula-3 report, Report #31, DEC Systems Research Center, August 1988.
- [Coquand Huet 85] T.Coquand, G.Huet: Constructions: a higher order proof system for mechanizing mathematics, Technical report 401, INRIA, May 1985.
- [Cook 87] W.Cook: A self-ish model of inheritance, manuscript, 1987.
- [Courcelle 83] B.Courcelle: Fundamental properties of infinite trees, Theoretical Computer Science, 25, pp. 95-169, 1983.
- [Dahl Nygaard 66] O.Dahl, K.Nygaard: Simula, an Algol-based simulation language, Comm. ACM, Vol 9, pp. 671-678, 1966.
- [Demers Donahue 79] A.Demers, J.Donahue: Revised Report on Russell, TR79-389, Computer Science Department, Cornell University, 1979.

- [Futatsugi Goguen Jouannaud Meseguer 85] K.Futatsugi, J.A.Goguen, J.P.Jouannaud, J.Meseguer: Principles of OBJ2, Proc. POPL 1985.
- [Girard 71] J-Y.Girard: Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types, Proceedings of the second Scandinavian logic symposium, J.E.Fenstad Ed. pp. 63-92, North-Holland, 1971.
- [Gordon Milner Wadsworth 79] M.J.Gordon, R.Milner, C.P.Wadsworth: Edinburgh LCF, Springer-Verlag Lecture Notes in Computer Science, n.78, 1979.
- [Harbison Steele 84] S.P.Harbison, G.L.Steele Jr.: C, a reference manual, Prentice Hall 1984.
- [Harper Milner Tofte 88] R.Harper, R.Milner, M.Tofte: The definition of Standard ML - Version 2, Report LFCS-88-62, Dept. of Computer Science, University of Edinburgh, 1988.
- [Hyland Pitts 87] J.M.E.Hyland, A.M.Pitts: The theory of constructions: categorical semantics and topos-theoretic models, in Categories in Computer Science and Logic (Proc. Boulder '87), Contemporary Math., Amer. Math. Soc., Providence RI.
- [Krasner 83] G.Krasner(Ed.): Smalltalk-80. Bits of history, words of advice, Addison-Wesley, 1983.
- [Landin 66] P.J.Landin: The next 700 programming languages, Comm ACM, Vol. 9, No. 3, 1966, pp. 157-166.
- [Liskov et al. 77] B.H.Liskov et al.: Abstraction Mechanisms in CLU, Comm ACM 20,8, 1977.
- [Liskov Guttag 86] B.H.Liskov, J.Guttag: Abstraction and specification in program development, MIT Press, Cambridge, MA, 1986.
- [Lucassen Gifford 88] J.M.Lucassen, D.K.Gifford: Polymorphic Effect Systems, Proc. POPL '88.
- [MacQueen 84] D.B.MacQueen: Modules for Standard ML, Proc. Symposium on Lisp and Functional Programming, Austin, Texas, August 6-8 1984, pp 198-207. ACM, New York.
- [MacQueen Plotkin Sethi 86] D.B.MacQueen, G.D.Plotkin, R.Sethi: An ideal model for recursive polymorphic types, Information and Control 71, pp. 95-130, 1986.
- [Martin-Löf 80] P.Martin-Löf, Intuitionistic type theory, Notes by Giovanni Sambin of a series of lectures given at the University of Padova, Italy, June 1980.
- [Milner 84] R.Milner: A proposal for Standard ML, Proc. Symposium on Lisp and Functional Programming, Austin, Texas, August 6-8 1984, pp. 184-197. ACM, New York.
- [Mitchell Plotkin 85] J.C.Mitchell, G.D.Plotkin: Abstract types have existential type, Proc. POPL 1985.
- [Mitchell Maybury Sweet 79] J.G.Mitchell, W.Maybury, R.Sweet: Mesa language manual, Xerox PARC CSL-79-3, April 1979.

- [Parnas 72] D.L.Parnas: On the criteria to be used in decomposing systems into modules, Communications of the ACM, Vol. 15, no. 12, pp. 1053-1058, December 1972.
- [Plotkin 81] G.D.Plotkin: A structural approach to operational semantics, Report DAIMI FN 19, Computer Science Department, Aarhus University, 1981.
- [Reynolds 74] J.C.Reynolds: Towards a theory of type structure, in Colloquium sur la programmation pp. 408-423, Springer-Verlag Lecture Notes in Computer Science, n.19, 1974.
- [Schaffert Cooper Bullis Kilian Wilpolt 86] C.Schaffert, T.Cooper, B.Bullis, M.Kilian, C.Wilpolt: An introduction to Trellis/Owl, Proc. OOPSLA'86.
- [Strachey 67] C.Strachey: Fundamental concepts in programming languages, lecture notes for the International Summer School in Computer Programming, Copenhagen, August 1967.
- [Stroustrup 86] B.Stroustrup: The C++ programming language, Addison-Wesley 1986.
- [Turner 85] D.A.Turner: Miranda: a non-strict functional language with polymorphic types, in Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science No. 201, Springer-Verlag , 1985.
- [Wirth 83] N.Wirth: Programming in Modula-2, Texts and Monographs in Computer Science, Springer-Verlag 1983.
- [Wirth 87] N.Wirth: From Modula to Oberon, and the programming language Oberon, Report 82, Institut für Informatik, ETH Zürich, 1987.